

Grails技术精解与Web开发实践

宋友 梁士兴 黄璐 王鑫 等编著

清华大学出版社

Grails 技术精解与 Web 开发实践

宋友　梁士兴　黄璐　王鑫　等编著

清 华 大 学 出 版 社

内 容 简 介

Grails 是继 J2EE、PHP、ROR 等技术之后又一个成功的 Web 框架。本书系统讲解了使用 Grails 技术快速进行 Web 开发的知识。本书内容分为四大部分，第一部分介绍了 Grails 必备的基础知识，包括环境配置、Groovy 语言基础、HelloWorld 程序开发实例等；第二部分以迭代的方式，设计并逐步完善了一个购物车应用，介绍了 Grails 基础知识；第三部分对 Grails 各个部分的细节进行了深入讨论；第四部分对 Grails 的实现原理进行了剖析，通过分析 Grails 的源码，帮助读者体会出 Grails 如此神奇的奥妙所在。

本书可作为大学本专科软件工程等专业教材，其适用的课程可以为 Web 开发与实践、软件工程实践等。本书也可供 Web 开发与应用的工程技术人员和爱好者参考。

图书在版编目（CIP）数据

Grails 技术精解与 Web 开发实践 / 宋友等编著. —北京：清华大学出版社，2009.8

ISBN 978-7-302-20187-8

Ⅰ. G… Ⅱ. 宋… Ⅲ. 软件工具－程序设计 Ⅳ. TP311.56

中国版本图书馆 CIP 数据核字（2009）第 076421 号

责任编辑：夏兆彦
责任校对：徐俊伟
责任印制：
出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185×260　**印　张**：16.5　　**字　数**：408 千字
附光盘 1 张
版　　次：2009 年 8 月第 1 版　　**印　次**：2009 年 8 月第 1 次印刷
印　　数：1～ 000
定　　价：　　元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：010-62770177 转 3103　产品编号：031598-01

FOREWORD

推荐序

互联网的发展带来了新的科技革命，Web 2.0 的概念更是将网络技术和应用推向了新的高潮。在这个技术日新月异发展的年代，各种框架、各种概念、各种思想呈现出了百花齐放局面，虽然无比繁荣，却让人十分眼花缭乱。选择一种最适用的 Web 框架，可以给开发时间、开发成本、开发质量带来巨大的影响。

当今世界是一个创造奇迹的时代。J2EE 创造了一个奇迹，使得开发企业级的 Web 应用变得快速、可靠、经济，从而成为近年来的主流 Web 开发框架。RoR 的诞生创造了另一个奇迹，它的开发效率是 J2EE 的 5 倍以上，而且相比 PHP 技术，它有着更好的代码组织结构，更容易写出高质量的代码。

Grails 是继 RoR 之后的又一个奇迹：它不但具备 J2EE 和 RoR 的诸多优点，还解决了 RoR 不能有效适用于广大 Java 用户的一大难题。目前，Grails 已得到了业内广泛地认可。在 2008 年 11 月 11 日，以 Spring 框架而闻名于天下的 SpringSource 宣布收购主导 Grails 开发的 G2one 公司，并承诺在收购之后，会投入更大的力量去改进 Grails 并且对 Grails 的全球用户提供商业支持。正如著名的 ququjoy 的评论所述：“SpringSource 将 G2One 收购之后，以 Spring 为底层框架的 Grails 必将迎来自己的春天”。对于还在观望的用户，还犹豫什么呢？赶快加入到 Grails 的使用者队伍中吧，充分享受 Grails 带来的实用与便捷。

目前，市面上关于 J2EE 的书籍数不胜数。关于 RoR 的书籍也至少有数十本。然而，也许很快将引领潮流的 Grails 实在太新了，它的资料主要还在 Grails 官方网站上能获得。关于 Grails 的中文书籍仅有两本外版图书，并且介绍的是低版本的 Grails。这对于广大爱好 Web 开发、并渴望学习 Grails 这门新技术的读者来说，不能不说是一个遗憾。

可喜的是，这本书问世了，该书的作者们自己使用 Grails 开发了许多成功的应用。为了撰写该书，他们亲自编写了一个典型的网上购物车的 Web 应用实例。通过 Web 实例制作的示范和讲解，由浅入深地分析了 Grails 的神奇，实践性非常强。同时，本书也对 Grails 的源代码、基于 MVC 的 Web 开发原理进行了深入的分析，亦不失理论深度。该书的面市相信会为 Web 开发爱好者带来一个小的惊喜。

该书优美地结合了理论与实践。不失为一本优秀的教科书和程序员与项目经理的参考手册。值得一读。

孙 伟

北京航空航天大学软件学院院长

2009 年 4 月 5 日

FOREWORD

前言

目前，虽然 J2EE 仍然是企业级 Web 开发的主流，但越来越多的企业和个人在开发新项目时更倾向于尝试使用 RoR，因为用 RoR 开发 Web 应用，可以更快捷、更简单。仿佛在一夜之间，RoR 成为了 Web 框架领域内最耀眼的明星，RoR 的一些理念，如约定优于配置，已经成为了新框架设计的金科玉律。

然而，RoR 对于世界上数量最大的 Java 用户来说，并不足够友好。第一，Ruby 语法与 Java 类语言有较大区别，需要克服较大的障碍才能掌握。第二，历史遗留项目问题，Java 语言的广泛流行使得大量既有项目是用 Java 开发的，而且这些项目还需要被继续支持和扩展，扩展历史遗留项目的任务，RoR 显得无能为力。第三，性能问题，Ruby 语言相对于 Java，其执行性能较差。上面这些因素，都会制约 RoR 在实际项目中的使用。正是在这一背景下，Grails 诞生了。

Grails 的全称为 Groovy on Rails，是一种基于 Groovy 语言的 Rails 类框架。它继承了 RoR 的绝大多数优点，几乎和 RoR 一样简单。而对于 RoR 的不足，它有着明显的优势。第一，Groovy 语言运行于 JVM，可以透明地执行 Java 程序。Java 程序员可以迅速地掌握 Groovy，同时对基于 Java 的历史遗留项目，可以得到非常好的支持。第二，Groovy 语言的执行性能虽然不及 Java，仍然比 Ruby 有较大的优势。另一方面，Grails 虽然名为 Groovy on Rails，但实际上 Grails 的主要开发语言是 Java，Grails 框架有的绝大多数代码是用 Java 编写的，因而，Grails 的性能就更可以得到保障。

Grails 技术凭借它卓越的设计，给用户提供了一种高效、简单易用、一栈式 Web 开发的框架。利用 Grails 技术，可以以极低的学习代价，换来很高的开发效率。使用 Grails 开发 Web 应用，需要使用的编程语言是 Groovy 语言。Groovy 是运行于 JVM 之上的程序设计语言，近些年发展迅猛。对于 Java 用户，要想掌握 Groovy，是非常容易的，学习曲线也非常平缓，甚至不需要学习，仅凭借 Java 的基本功，就可以把 Grails 驾驭得非常好。因此，极力推荐正在使用 Java 开发 Web 应用的公司、学生、个人用户学习并使用 Grails。对于没有掌握 Java 技术，但很想学习 Web 开发技术的用户，那不妨就直接从 Grails 开始吧，Grails 非常容易掌握。使用 Grails，开发人员就可以更多地把精力花在业务上，而不是技术细节上。

一、阅读本书需要的预备知识

现代的 Web 应用通常会涉及到页面设计、数据存储等多个方面。Grails 将这方方面面的问题都考虑了进来，因而说 Grails 是一个一栈式的框架。本书预期的读者，应预备以下几方面的基础知识。

1．HTML 的基础知识。是的，无论如何，HTML 都是 Web 技术的基础，只有具备一定的 HTML 基础，才能设计并输出 Web 页面的基本元素。同时，应该对表单的提交（GET/POST）有所了解。当然，这部分内容非常容易掌握，通过百度 Google，可以轻松地找到许多有用的教程。

2．数据库的基础知识。Grails 对数据库的操作进行了封装和简化，但开发人员仍然有必要掌握一定的数据库知识，对数据库表、主键、外键、索引，有基本的了解。能够读写简单的 SQL 语句。只有这样，才能真切地体会到 Grails 与生俱来的快捷与便利，而且遇到问题才不会茫然。

3．Java 语言或 Groovy 语言，或者其他类似的面向对象程序语言。使用 Grails 进行开发，需要使用 Groovy 语言。掌握 Java，就意味着在相当程度上掌握了 Groovy。它们二者间比较重要的差异，本书中会进行必要的介绍。当然，对于不了解 Java 但至少掌握了一种编程语言的读者，学习 Groovy 也不会有任何困难，毕竟 Groovy 的设计思想就是设计一种比 Java 更简单易用的编程语言。

4．Hibernate 基础知识（非必需）。本书会对 Grails 的部分实现原理进行深入和分析，会涉及到许多 Hibernate 中的概念。读者如果只是希望用 Grails 开发自己的 Web 应用，则无需要求这一点。

二、关于本书的内容说明

本书的内容包括四大部分。从第一部分到第四部分，按知识的难度和深度，存在一定的递进关系。但各部分也相对独立。

第一部分是入门篇，介绍了 Grails 必备的基础知识，包括：环境配置、Groovy 语言基础、HelloWorld 程序开发实例，以及一些基本概念。对于熟悉 Java 编程的读者，可以快速浏览本部分。

第二部分共 7 章，以迭代增量的方式，完整地设计并开发了一个 Web 应用——网络购物车。该部分内容从最简单的环境准备开始，接下来包括数据创建、数据查询、数据维护与管理、系统后台管理、系统测试、系统部署等。该部分是一个逐步完善的过程，每开始新的一章，就是增加了一些新的功能和知识点。

学习完前两部分，可以掌握 Grails 主要的基础知识，并能开发一些简单的 Web 应用。这可以称得上是 Grails 开发的初级用户了。

第三部分共 6 章，对 Grails 的关键问题进行了比较深入的讨论，对相关的原理进行了介绍。该部分内容包括 Grails 的 MVC 原理、GSP 输出、Web Service 实现、Grails 插件使用等内容。本部分的内容有一定难度和深度，但这些知识也是 Web 开发的基本要素，要开

发真正实用的 Web 应用，掌握这些知识是必要的。

掌握了前三部分内容的读者，可以称得上是中级用户了，能够比较熟练地使用 Grails 开发 Web 应用。

第四部分是最后 5 章，其内容包括：介绍 Groovy 的高级特性、分析 Grails 的源码、讲解 Grails 的插件开发、阐述 Grails 的新特性等。该部分内容难度和深度都较大，是对 Grails 的一些原理进行深入分析。掌握该部分内容，有助于帮助读者体会出 Grails 之所以如此神奇的奥妙所在，在实践中能够通过灵活运用 Grails，并可能修改源码，进行更丰富的二次开发。

在实例分析和原理解析上，本书两者并重，它们分别占了约一半的篇幅。但两者并不是截然分离的，有的时候两者是有机地结合在一起。读者通本书实例的学习实践，可以依葫芦画瓢地开发自己的 Web 应用；通过品味 Grails 的原理，将能开发出更多、更复杂的 Web 应用。读完这本书，读者不仅仅能开发一个网上购物车，同样能开发自己的工作流、表单系统、等等。

由于 Grails 的知识很新，可参考的文献非常少，本书的知识大部分是作者从 Grails 的官方资料中获得的。本书是作者通过阅读和分析 Grails 的源码，亲自开发 Web 应用，编写了较多的实际代码，然后再总结自己从事 Web 开发的一些经验而完成的。本书原创内容较多，因而疏漏之处可能不少，诚恳地请读者批评指正。

三、代码的阅读说明

本书的代码可从网络下载，网址为 www.tup.tsinghua.edu.cn，包括了网络购物车的完整代码。

在本书中，因为购物车的开发是增量进行的，所以随着每一章的代码逐步添加，最后才形成了完整的应用代码。但读者在阅读过程中，阅读并实践到某些章节时，因为还不涉及到后面章节的内容，所以将会发现自己按书上步骤介绍而写出来的代码跟我们提供代码的内容并不一致。

但我们不打算把每一章的代码都单独放在一个文件夹里，写成一个独立的应用。一方面，这样会破坏该应用的完整性；另一方面，也是更重要的，读者在阅读每一章时，可能会不假思索地把我们的代码复制到开发环境里，顺利地让程序运行起来，自己却可能并不明白代码说的是什么。

读者在阅读并实践本书的应用实例时，我们鼓励读者按书上描述的步骤，尽量自己亲自输入相关的代码，并运行程序分析结果。如果有些例子的代码的确非常多，可以从我们提供的代码里摘录部分核心内容。

另外，在分析实例时，有些知识点的代码也比较长，我们在书里也没有全部把代码列出来，读者可以参阅网络文件里的相关内容。

四、写作小札

在 2008 年 4 月初时，我们刚使用 Grails 开发了一个成功的商业应用。在开发该应用的过程中，我们深刻感受到 Grails 中文资料在国内的匮乏，而且原创的 Grails 中文书籍在

国内还没有。另外，市面关于 Grails 的书籍中，基本上都是以 Grails 0.3 为基础来介绍的，但当时主流应用的是 Grails 1.03，两者的差距较大。这给广大 Web 爱好者学习和应用 Grails 带来了困难。于是，我们就萌发了写作此书的想法。

2008 年五一后，该写作计划开始启动，并且该计划得到了清华大学出版社和北航软件学院的大力支持。

我们原计划用大约半年时间完成该书的写作，但事实上花了近一年时间才完稿。实际写作过程中，开发一个完整的 Web 应用实例并不是难事。但在分析 Grails 的原理时，要把 Grails 之所以神奇的本质阐述清楚（例如 Grails 如何通过 Hibernate 进行数据库操作、如何通过 Spring 引入外部的组件、等等），则并不容易，主要是这方面的参考资料非常少，作者不得不花大量的功夫来仔细研读并分析 Grails 的源代码，然后用通俗的语言，并设计一些例程进行解释。

本书的大部分内容是在 2008 年完成的，这其中经历国人大悲的 5.12 地震和大喜的北京奥运会，因此，我们的写作心情也受到巨大的影响，前半年时间是在心情很不平静的情况下完成写作的。整个暑假，我们每天都在坚持写作，当然，期间有伴随着边看奥运会边写作的一段快乐日子。在此期间，我们也承接了一些用 Grails 开发的 Web 应用小项目，当然，那时我们使用 Grails 开发的速度已非常快，效益比率也非常可观。开发这些应用，对于我们进一步理解 Grails，从而更好地写作该书提供了许多有意义的启发。

我们写作的基础曾是 Grails 1.03，但 Grail 的发展很快，寒假前夕，我们的书稿快完时，Grails 1.04 正式版出来了，我们不得不把整本书的相关内容都换成 1.04 版的内容。当时，1.1 版的测试版也有了，于是我们还特别地介绍了部分新版本的新特性。但很遗憾，刚完稿没多久，1.1 的正式版又出来了。我们深刻理解，计算机领域，技术的更新速度永远都比知识传播得要快，因此我们知道，我们难以保证今天出的书里面的内容一定反映今天最新的事情，虽然我们有时可以预测今天或明天将要发生的事情。该书目前以 Grails 1.04 为基础进行介绍，这对于读者学习和使用 1.1 版而言，应该说没有影响。不过我们希望，每隔一段时间后，能不断有新结合 Grails 新版本、反映 Grails 新特性的书籍出现。下一个作者，也许就是你——今天的读者。

该书写完后，本书的作者之一，梁士兴也顺利硕士毕业了。在进行该书的写作过程中，他还完成了自己的学位论文写作、论文答辩、就业等事情，并且，写作这本书也有力地促进了他在学业和就业上的进步。最终，他以优良的毕业成绩通过答辩，并得到了自己满意的工作——获得了 IBM 的 offer。于是，他得到一个感悟：读一本好书、学习一门技术、开发一个项目、撰写一本书和文章，其收获可能是自己也无法预知的一个惊喜。

五、致谢

在本书近一年的写作过程，得到了广大老师、朋友的热心支持。

感谢北京航空航天大学软件学院的领导和老师对该书撰写工作的支持，并提供了良好的写作条件，该项目得到了北京市教育委员会第二类特色专业建设项目支持。

感谢曹媛、李翔昊、吕经纬、李宇杰、刘其帅、宋恒、赖景愚等同学作为该书的第一批阅读者和实践者，并对该书的内容提出了许多宝贵的修改意见。

在此，谨对关心和帮助本书完成的所有老师、同学和朋友，以及相关单位致以诚挚的感谢。

宋友　　北京航空航天大学软件学院

梁士兴　IBM 中国开发中心

黄璐　　IBM 中国开发中心

王鑫　　IBM 中国开发中心

2009 年 4 月　北京

CONTENTS

目录

第二篇 实际应用

第三篇 深入了解 Grails

第 1 章 导论

1.1 RoR 的革命与 Web 开发的新时代

随着互联网的普及和发展，绝大多数计算机应用被设计为基于 Web 技术的 Web 应用。Web 应用有着发布快捷、使用简便、对客户端要求低等优势。通过 Web 技术，可以让企业更容易发布服务，也可以让用户更容易享受信息时代的便捷。Web 的优势导致了 Web 开发的需求呈现出爆炸式的增长，进一步则对 Web 开发技术提出了更高的要求。

RoR（Ruby on Rails）的诞生就创造了一个这样的奇迹，它在短短的几年间，在无数如 JSP、PHP 等元老级的 Web 技术面前脱颖而出，并且给了整个 J2EE 世界以极大的震撼！正如 RoR 所标榜的，它的开发效率是 J2EE 的 5 倍以上。这样巨大的差异不可能不引起 Java 世界的反思。

首先，为什么 Rails 这样成功的开发框架会诞生于 Ruby 语言，而且在其出现之后，Java 世界并没有产生能与之抗衡或者模仿它的框架呢？其次，Java 在企业级开发中一直追求的重量级的大而全，真的是用户的最想要吗？

带着这样的几个问题，这里不妨简单分析一下 RoR 所取得的成功。

首先 RoR 的优雅与快捷，相当程度上源于动态脚本语言 Ruby 的“元编程”技术。

所谓动态语言，一般有两层含义：动态类型和动态结构。RoR 的基础——Ruby 语言，同时符合上面的两种特性。支持动态类型，指只在运行时确定 Ruby 的变量类型信息；支持动态结构，指利用“元编程”技术，在程序运行时，对 Ruby 类的属性和方法进行动态增减。“元编程”的技术可以用于实现更灵活的 Web 框架，它是 Java 等静态语言所不具备的特性，因而很遗憾，Java 不可能实现出 Rails 这样的框架。

谈概念，总是让人昏昏欲睡。不妨看看例子。假设数据库的 Goods 表包含 name 和 title 字段，那么针对这两个字段进行联合查询的代码如下：

```
def goodsList = Goods.findAllByNameAndTitle(name, title)
```

用传统方法（如 Java）设计框架，想实现提供类似 findAllByNameAndTitle 这样的方法，是不现实的。因为仅 name 和 title 就会有“name and title”和“name or title”两种条件组合（考虑到 name 和 title 前后位置替换，虽然逻辑相同，但对应的函数名不同，所以应该是排列），再加上其他属性，产生的查询和排序会呈现出爆炸式的增长，再聪明的框架也不可能预先准备好应付所有的情况。通常需要用户自己去实现相关的查询逻辑，即自己

编程实现 findAllByNameAndTitle。显然，利用 Ruby 的“元编程”技术动态添加方法，在框架级别实现支持这样的动态查询，无疑会使开发变得简单很多。而 Java 这样的静态语言则很难实现这样的功能。因此，可以这样理解，Rails 利用了 Java 不支持的动态脚本语言的特性，实现了许多更有利于减轻用户开发劳动的功能，从而让用户觉得 Rails 更简单快捷。Rails 正是因为这样而取得了成功。

RoR 的设计指导思想就是简化开发过程。传统的 Java 开发方式，其设计显得过于学术化，而忽略了用户的感受。典型的如 SSH（Struts + Spring + Hibernate）的开发方式，为了追求灵活性，系统的各层都可以替换成其他的框架。虽然灵活，但对简化开发带来的好处却非常有限。RoR 则设计得非常实用，提供一个一栈式的解决方案，各层的设计都只有一个目标，那就是简单！

虽然 RoR 比 J2EE 简单，但 RoR 同样有着良好的组织结构。PHP 也是十分简单易用的 Web 开发技术，与 PHP 相比，Rails 最大的优势就在于它的组织结构更加清晰。RoR 是一个 MVC（Model-View-Control，模型-视图-控制器）的 Web 框架，它在工程中预先定义了若干的文件夹，分别用于存放模型-视图-控制器，其结构非常清晰明了。这体现 RoR 的另一个理念，即约定优于配置：约定不同的文件夹中存放特定类型的文件，而无需进行配置。

1.2　RoR 并不完美

RoR 凭借着它极高的开发效率和飞速增长的用户数量，毫无疑问地成为了现如今最成功的 Web 开发技术之一。然而，RoR 并不完美，它在几个方面还存在着一定的不足。

1.2.1　Ruby 语言方面的不足

虽然 Ruby 也是一种优秀的脚本语言，但它存在着执行性能差、GC（Garbage Collection，垃圾回收）缺陷等不足。

Ruby 的语法相对 Java 有较大的区别，对 Java 的程序员而言，需要学习一门全新的编程语言，对编程习惯有较大的冲击。

基于 Ruby 的开源项目较少，一些常见的开发过程中的问题，如工作流、全文检索等，都还缺乏成熟的解决方案。

1.2.2　对历史遗留项目的支持较为困难

Java 仍然是现在最流行的编程语言，在企业级别，有大量基于 J2EE 技术的历史遗留项目。对于这样的企业，如果采用 Ruby on Rails 开发新系统，则意味着难以实现对历史遗留项目的支持。

1.3 Grails 的诞生解决了一些遗憾

Grails 的全称为 Groovy on Rails，是一种基于 Groovy 语言的 Rails 类框架。它继承了 RoR 的绝大多数优点：Grails 几乎和 RoR 一样简单，甚至在某些方面，比 RoR 更简单强大（如数据库的查询访问）；Grails 同样也遵循约定优于配置的指导思想。除此之外，它还有如下几个不同于 RoR 的特点。

1.3.1 Groovy 语言

Groovy 是与 Ruby 不同的脚本语言，虽然 Groovy 也是脚本语言， 但它是以 Java 字节码的方式运行在 Java 虚拟机之上的。这样有两方面好处：一方面，可以有较好的性能；另一方面，Groovy 可以在语言级别透明地调用 Java 程序。因而，使用 Groovy 可以良好地兼容历史遗留项目，并且能够运行在成熟可靠的 J2EE Server 之上。

Groovy 有着介于 Java 和 Python 之间的语法风格，灵活不失稳重。给 Java 程序员降低了学习的门槛。

1.3.2 Grails 站在了巨人的肩膀之上

Grails 在实现过程中，使用了大量成熟可靠并为世人所广为认可的 Java 开源项目。使用了 Spring MVC 框架作为底层，实现对 Web MVC 的支持；使用了 Hibernate 作为底层，实现 GORM（GORM 是 Grails 提供的对象关系映射框架）；使用 Spring IoC 容器，实现对系统各个模块的组织与管理；使用 SiteMesh，实现对页面布局（layout）的统一和简化。

有了这些成熟可靠的项目作为基础，Grails 的可靠性得到了保证，复杂性得到了降低。从而使得 Grails 可以在更高的起点上，更有效地解决实际开发过程中的问题。

1.3.3 Grails 有良好的扩展性

Grails 采用了插件化的设计架构，不仅现有的功能是通过插件实现的（系统插件），还可以通过插件的方式，加入更多新功能，或者对其他的一些流行框架（如 Spring Security、JQuery 等）进行整合。

目前 Grails 官方网站上，提供了近百个插件，用以解决方方面面的问题。有效地利用这些插件，可以显著地提高开发效率和降低开发的难度。

1.4 对 Grails 的一些误解

作为新兴事物的 Grails，人们还对其存在一些误解[1]。

（1）Grails 还不够成熟。

事实上，已经有越来越多的组织和个人选用 Grails 技术开发网站，Grails 官方站点已经列出了上百个使用 Grails 技术的成功商业案例[2]。此外，由于 Grails 本身是基于 Hibernate、Spring 和 SiteMesh 这些成熟完善的框架而构建的，因而无需担心它的成熟程度。

（2）Grails 是否只是 Rails 的一个克隆产物。

Ruby on Rails 引入了不少非常好的主意，Grails 将其中的一部分应用到了 Groovy/Java 世界中，并且还加入了许多 Ruby 中并不存在的特性和概念，所有这些东西都是以一种对 Groovy 和 Java 程序员有意义的方式展现出来的。

（3）有了 JRuby on Rails 之后，谁还要 Grails 呢？

Grails 的目标与 JRuby on Rails 大为不同。它不是为了在 Groovy 语言上实现一个 Rails 的移植版本，而是将业界最为强悍的组件（如 Spring、Hibernate、Quartz、Compass 和 SiteMesh 等）以最佳方式组合起来的一个实践，并通过采纳无配置规约使它们符合“不重复（Don't Repeat Yourself，DRY）”的原则。Grails 没有重复造轮子，而且，由于 Grails 内核的绝大部分代码都是以 Java 编写的，它也显得更加强壮、稳定和高效。事实上，从内核角度看 Grails 应用，只相当于是一个 Spring MVC 应用，因而可以被部署到所有的主流容器之上，包括大型的商业容器，如 WebLogic、WebSphere 和 Oracle AS。

1.5 本书的使用说明

本书统一对代码和控制台输入输出使用 Consolas 字体，这是一种等宽的字体。具体显示效果如下：

```
def goodsList = Goods.findAllByNameAndTitle(name, title)
```

对于比较重要的内容，会标记为粗体或粗斜体，如以下代码所示：

```
def goodsList = Goods.findAllByNameAndTitle(name, title)
println goodsList
goodsList = Goods.list()
println goodsList
```

此外，对于通过控制台输入的命令，会使用“>”开头，如：

```
>cd Helloworld
```

1.6 本章小结

本章对 RoR 的成功进行了简单的分析，指出了它的一些不足之处。Grails 框架能有效地弥补 RoR 的这些不足之处，因而更适合作为 Java 程序员首选的 Web 开发技术。本章的最后对本书的字体、格式进行了简要说明。

第一篇

入　门　篇

这部分包含两章，这两章将介绍一些比较基础的 Grails 和 Groovy 的知识，包括：如何安装 Grails，如何用 Grails 创建一个 Web 应用程序，Groovy 与 Java 的简单对比，等等。

第2章 Hello Grails

2.1 Grails 的安装

由于 Groovy 是运行在 JVM 之上的语言，所以在安装 Grails 前，应先确保本机已经安装了 JDK，并配置它的环境变量。这里要求使用 JDK5.0 以上的版本，推荐使用 JDK6。如果没有安装 JDK，可以到 SUN 的主页 http://java.sun.com 下载，并完成其环境变量的配置。若已熟悉并安装了 JDK，可跳过 2.1.1 小节，直接从 2.1.2 小节开始阅读。

2.1.1 JDK 的安装与配置

JDK 的安装并不复杂，只需要按照向导的提示，一步步地进行即可。但安装完成后需要配置 JDK 的环境变量，具体步骤如下所示。

（1）创建 JAVA_HOME 为 JDK 的安装目录，如图 2-1 所示。

（2）创建 CLASSPATH 为.\;%JAVA_HOME%\lib\tools.jar;%JAVA_HOME%\lib\dt.jar，如图 2-2 所示。

（3）更新 Path，加入%JAVA_HOME%\bin，如图 2-3 所示。



图 2-1　将 JDK 安装目录设为 JAVA_HOME

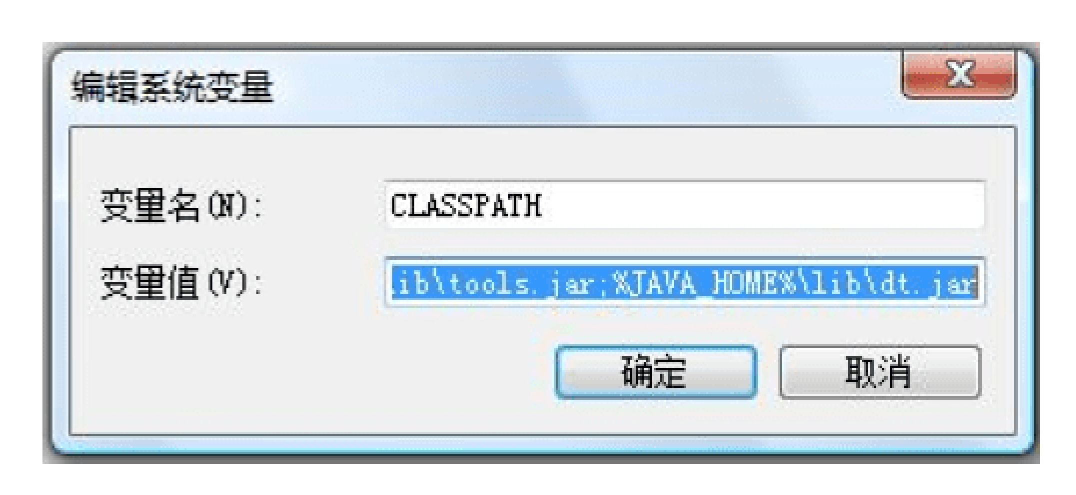


图 2-2　设置 CLASSPATH



图 2-3　将“%JAVA_HOME%\bin”加入 Path

配置完环境变量后，打开一个控制台，输入命令 javac -version，如果看到如下 Java 版本号（version）的提示，说明 JDK 已经安装配置成功。

```
>javac -version
javac 1.6.0_06
```

2.1.2　Grails 的安装

安装 Grails 之前，首先要下载它的程序包。可以在 Grails 的官方主页 http://www.grails.org 下载到。读者可根据需要，选择下载不含源程序的 binary 版，或下载含源码的 Source 版本[1]。下载完毕，将其解压到本机的某一路径下，如：“D:\”。

接下来，需要配置 Grails 的环境变量。将 Grails 解压后的路径配置为 GRAILS_HOME，如图 2-4 所示。



图 2-4　将 Grails 的解压路径设为 GRAILS_HOME

然后将%GRAILS_HOME%\bin 添加到系统环境变量的 Path 中，如图 2-5 所示。



图 2-5　将“%GRAILS_HOME%\bin”加入 Path

[1] 本书第四部分的章节会涉及到分析 Grails 实现原理，需要阅读 Grails 的源码。

至此，安装和配置就这样完成了。打开控制台，输入命令 grails，如果看到如下提示，就说明安装成功了：

```
>grails

Welcome to Grails 1.0.4 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: D:\grails-1.0.4
No script name specified. Use 'grails help' for more info or 'grails
interactive' to enter interactive mode
```

如果出现了报错消息，应检查 JDK 和 Grails 的环境变量是否正常。Linux 下的安装方法和 Windows 下基本一致，也需要在解压完成后配置环境变量。但倘若在 Linux 下解压完成后无法运行 grails 命令，可能还需要检查一下 bin 目录下的 grails 文件是否拥有执行权限。

好了，Grails 的安装已经完成了，难以置信的简单，不是吗？接下来，好好地去享受一下使用 Grails 进行开发的乐趣。

2.2 创建 Grails 工程

安装成功后，在控制台输入命令 grails help，会得到一串详细的命令列表，如图 2-6 所示。

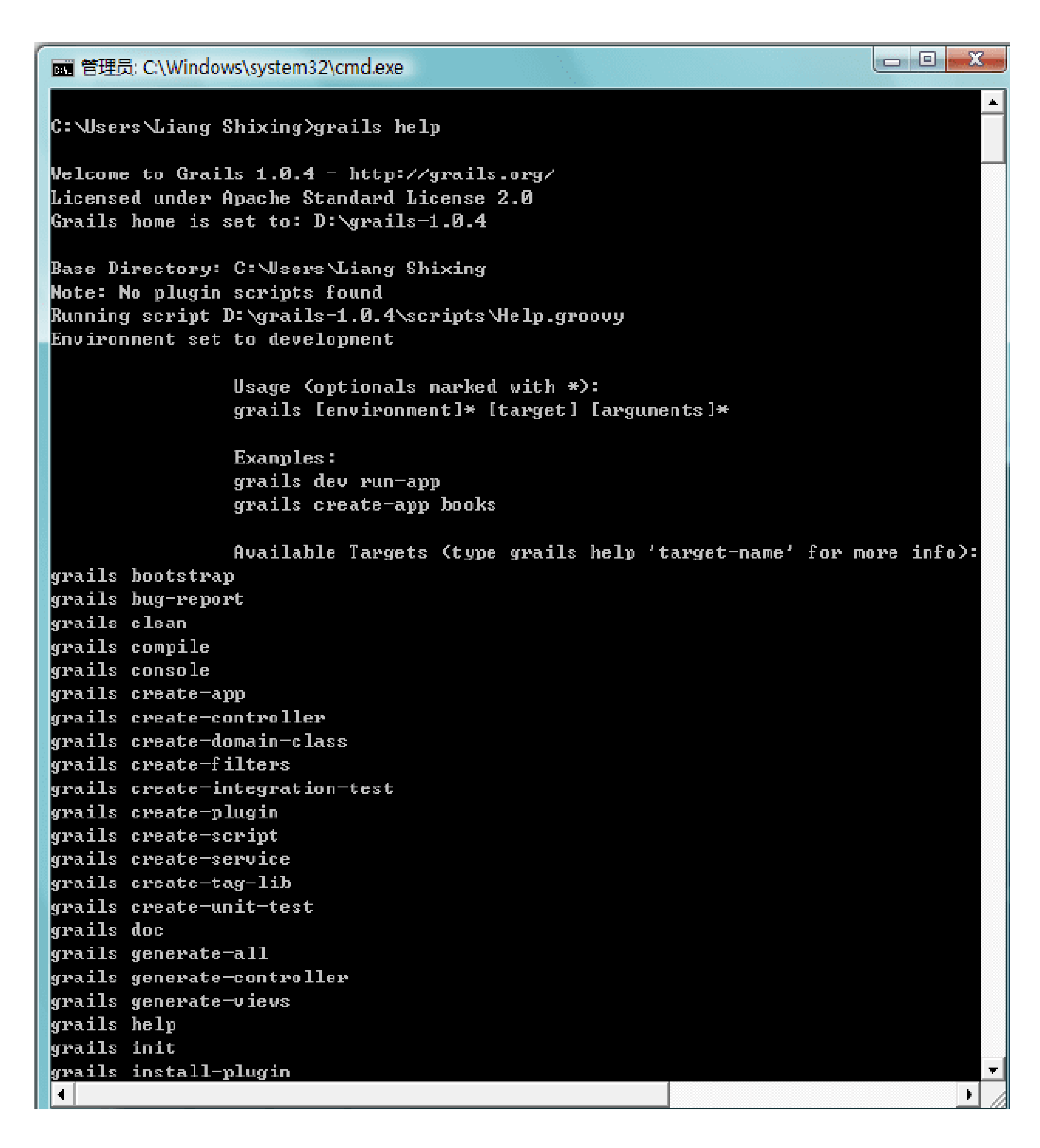


图 2-6　Grails 的帮助信息

这些命令将为开发人员的开发、测试、部署提供巨大的帮助，在后面的章节里会有更详细的介绍。这里先使用一个名为 create-app 的命令，去创建工程。

```
>grails create-app

Welcome to Grails 1.0.4 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: D:\grails-1.0.4

Base Directory: D:\workspace
Note: No plugin scripts found
Running script D:\grails-1.0.4\scripts\CreateApp.groovy
Environment set to development
Application name not specified. Please enter:
```

最后一行 Grails 提示输入程序名称。若输入 HelloGrails 并按 Enter 键，就会发现 Grails 创建了一个工程，该工程有比较复杂的目录结构和一些初始的文件：

```
[mkdir] Created dir: D:\workspace\HelloGrails\src
[mkdir] Created dir: D:\workspace\HelloGrails\src\java
[mkdir] Created dir: D:\workspace\HelloGrails\src\groovy
[mkdir] Created dir: D:\workspace\HelloGrails\grails-app
[mkdir] Created dir: D:\workspace\HelloGrails\grails-app\controllers
[mkdir] Created dir: D:\workspace\HelloGrails\grails-app\services
[mkdir] Created dir: D:\workspace\HelloGrails\grails-app\domain
[mkdir] Created dir: D:\workspace\HelloGrails\grails-app\taglib
[mkdir] Created dir: D:\workspace\HelloGrails\grails-app\utils
[mkdir] Created dir: D:\workspace\HelloGrails\grails-app\views
[mkdir] Created dir: D:\workspace\HelloGrails\grails-app\views\layouts
[mkdir] Created dir: D:\workspace\HelloGrails\grails-app\i18n
[mkdir] Created dir: D:\workspace\HelloGrails\grails-app\conf
[mkdir] Created dir: D:\workspace\HelloGrails\test
[mkdir] Created dir: D:\workspace\HelloGrails\test\unit
[mkdir] Created dir: D:\workspace\HelloGrails\test\integration
[mkdir] Created dir: D:\workspace\HelloGrails\scripts
[mkdir] Created dir: D:\workspace\HelloGrails\web-app
[mkdir] Created dir: D:\workspace\HelloGrails\web-app\js
[mkdir] Created dir: D:\workspace\HelloGrails\web-app\css
[mkdir] Created dir: D:\workspace\HelloGrails\web-app\images
[mkdir] Created dir: D:\workspace\HelloGrails\web-app\META-INF
[mkdir] Created dir: D:\workspace\HelloGrails\lib
[mkdir] Created dir: D:\workspace\HelloGrails\grails-app\conf\spring
[mkdir] Created dir: D:\workspace\HelloGrails\grails-app\conf\hibernate
[propertyfile] Creating new property file: D:\workspace\HelloGrails\
application.properties
 [copy] Copying 2 files to D:\workspace\HelloGrails
```

```
[copy] Copied 1 empty directory to 1 empty directory under
 D:\workspace\HelloGrails
 [copy] Copying 2 files to D:\workspace\HelloGrails\web-app\WEB-INF
 [copy] Copying 5 files to D:\workspace\HelloGrails\web-app\WEB-INF\tld
 [copy] Copying 28 files to D:\workspace\HelloGrails\web-app
 [copy] Copying 18 files to D:\workspace\HelloGrails\grails-app
 [copy] Copying 1 file to D:\workspace\HelloGrails
 [copy] Copying 1 file to D:\workspace\HelloGrails
 [copy] Copying 1 file to D:\workspace\HelloGrails
 [copy] Copying 1 file to D:\workspace\HelloGrails
[propertyfile] Updating property file: D:\workspace\HelloGrails\
application.properties
Created Grails Application at D:\workspace/HelloGrails
```

Grails 创建的 HelloGrails 工程所包含目录结构的含义，如表 2-1 所示。

表 2-1　Grails 生成的项目目录结构

目录	说明
HelloGrails	工程根目录
grails-app	
conf	存放配置信息，包含数据源、应用程序启动时自动执行的类 ApplicationBootStrap.groovy, Url 映射配置
spring	存放可选的 Spring 配置文件
hibernate	存放可选的 Hibernate 配置文件
controller	存放控制器（MVC 的 C）
domain	存放域类（MVC 的 M）
i18n	存放国际化资源文件
services	存放 service 类
taglib	存放标签库类
views	存放 GSP 页面（MVC 的 V，每个控制器对应一个文件夹并存放在 views 中，每个文件夹中会有多个 GSP 页面）
layouts	存放布局模板
utils	存放工具方法类
test	
unit	存放单元测试代码
integration	存放集成测试代码
lib	存放其他 jar 包（如 JDBC 驱动等）
src	
java	存放 Java 源程序
groovy	存放 Groovy 源程序
web-app	
css	存放 CSS 样式表
images	存放图片文件
js	存放 JavaScript 文件
WEB-INF	存放部署相关的文件
index.gsp	应用程序默认的首页

这里读者可以体会到一点，Grails 已经帮助开发人员设计好了很多东西，这是它所推

崇的约定胜于配置的思想。这可以让开发人员把更多精力集中在业务逻辑上，而不要为那些繁琐的项目配置浪费时间。

2.3 Grails 的 MVC 架构

Grails 学习了 Rails，也采用了 MVC 架构（见图 2-7）。在 Grails 中，“M”指的是 Domain 类，可以简单地将这个 Domain 理解为数据库里的一张表，一个 Domain 的实例则对应为该表的一条记录。通过操纵 Domain 类的实例，就可以实现对数据库进行增删查改操作。控制器“C”起到的是一个桥梁的作用，它能够接收用户提交的请求，它可以调用 M 获取数据并把数据传递给视图“V”。视图的作用是输出页面，Grails 中的页面技术，使用的是与 JSP 非常相似但更加简单易用的 GSP 技术，可以使用网页编辑器进行编辑设计。

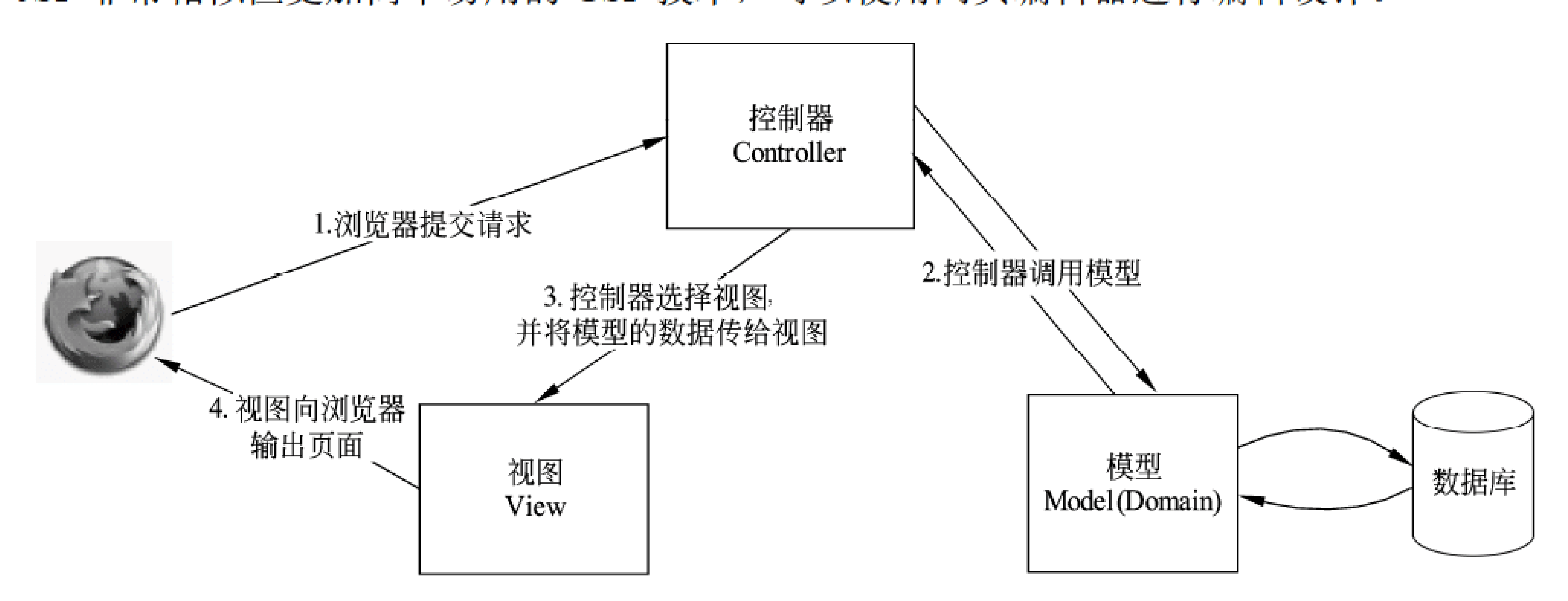


图 2-7 MVC 架构的 Grails 执行流程

这里的控制器和视图是一对多的关系，也就是说，一个控制器可能对应多个 GSP 页面。每个控制器都在 views 文件夹中对应一个同名的文件夹，而在这个文件夹中存放的就是它对应的 GSP 页面。

举个简单的例子，这里创建一个控制器。用 Grails 的 create-controller 命令创建一个控制器。该控制器命名为 hello。

```
>grails create-controller hello
Welcome to Grails 1.0.4 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: D:\grails-1.0.4

Base Directory: D:\workspace\HelloGrails
Note: No plugin scripts found
Running script D:\grails-1.0.4\scripts\CreateController.groovy
Environment set to development
    [copy] Copying 1 file to D:\workspace\HelloGrails\grails-app\controllers
```

```
    Created Controller for Hello
    [mkdir] Created dir: D:\workspace\HelloGrails\grails-app\views\hello
    [copy] Copying 1 file to D:\workspace\HelloGrails\test\integration
Created ControllerTests for Hello
```

可以看到 Grails 创建了控制器类，在 views 创建了名为 hello 的文件夹，还在 test 中创建了默认的测试程序。用任意的文本编辑器打开 grails-app\controller\HelloController.groovy 文件，可以看到如下内容：

```
class HelloController {
    def index = { }
}
```

这里的 def index = {} 是 Groovy 不同于 Java 的一种语法结构，叫做闭包[1]。闭包是一段代码的集合，与 Java 的方法类似，可以调用。在 Controller 中，闭包有了新的含义，叫做 action。Grails 会根据请求的 URL 决定调用哪个 action。对上面的代码进行简单的修改如下：

```
class HelloController {
    def say={
        render('Hello World! Hello Grails!')
    }
}
```

然后执行 Grails 的 run-app 命令来运行程序。

```
>grails run-app
```

当程序运行完成后，根据控制台输出的提示，打开任意的浏览器，访问地址 http://localhost:8080/HelloGrails/hello/say，可看到如图 2-8 所示页面。

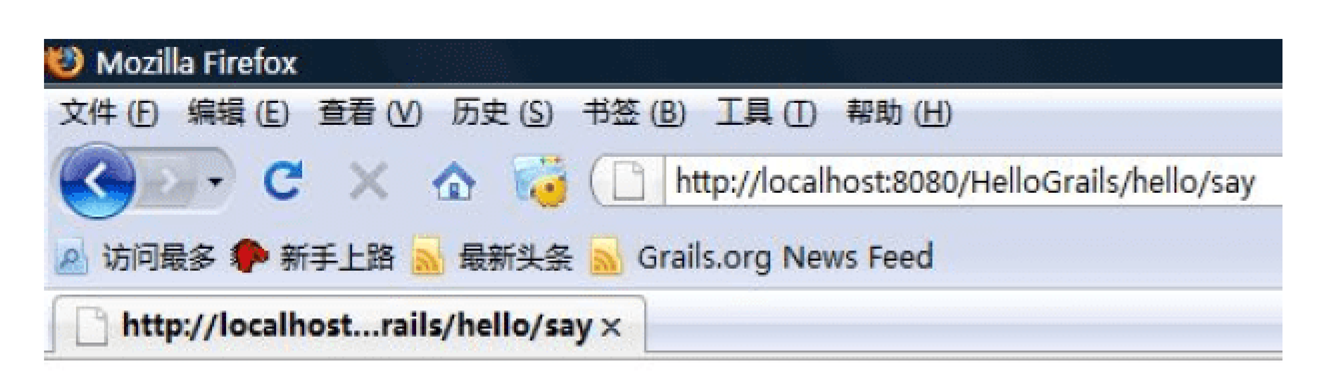


图 2-8　页面显示效果

不要小看这个例子，它可以帮助理解 Grails 的 URL 原理。一个典型的 Grails 的 URL 表现为如下样式：

```
http://主机名:端口/项目名称/控制器名/action 名/ID 或其他参数
```

于是http://localhost:8080/HelloGrails/hello/say就表示访问 HelloGrails 项目的 hello 控制

[1] 关于 Groovy 语法的介绍可以参考下一章。

器的 say action。不过心细的读者可能会问，View 的作用是怎么体现的呢？本例直接使用 render 方法输出页面，因此其中并没有体现出 View 的作用。下面把 View 加入到例子中，在 grails-app\views\hello 文件夹中创建一个名为 say.gsp 的文本文件。输入如下内容：

```
<html>
<head>
    <title>hello</title>
</head>
<body>
    Hello ${text1}! Hello ${text2}!
</body>
</html>
```

然后修改 HelloContoller 的内容如下：

```
class HelloController {
    def say = {
    return [text1:'World', text2:'Grails']
    }
}
```

刷新页面（无需重启 Grails，即重新运行 run-app 命令），就可以看到实现效果了，如图 2-9 所示。

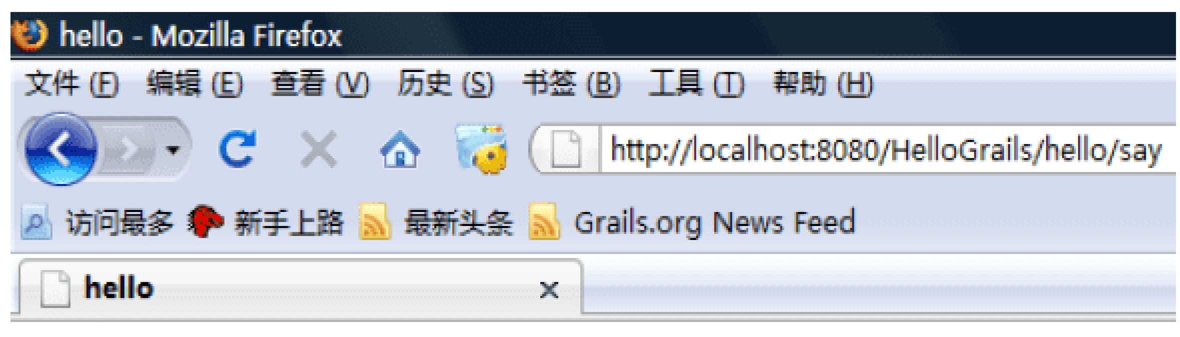


图 2-9 使用 GSP 输出的页面显示效果

虽然看起来和之前的效果基本上是一样的，但这一次使用了 View（GSP）来输出页面内容。这里有两个地方需要重点关注一下：一个是 Controller 向 GSP 传递数据的方式，Controller 是通过 action 的返回值[1]向 GSP 传递数据的，这个返回值是一个 Map[2]，Map 的 key 就成为了 GSP 页面中访问数据的变量名；另一个重要的地方是 Controller 如何对 GSP 页面进行选取，默认情况下，Controller 会自动选择与 action 同名的 GSP 去执行页面输出。

现在，HelloGrails 里已经包含了控制器与视图。下面，我们把模型引入进来，如图 2-10 所示。

[1] 在 Groovy 中，如果方法或闭包的最后一行是 return 语句，则 return 可以省略。

[2] 详见下一章对 Groovy 的集合、Map 的介绍。

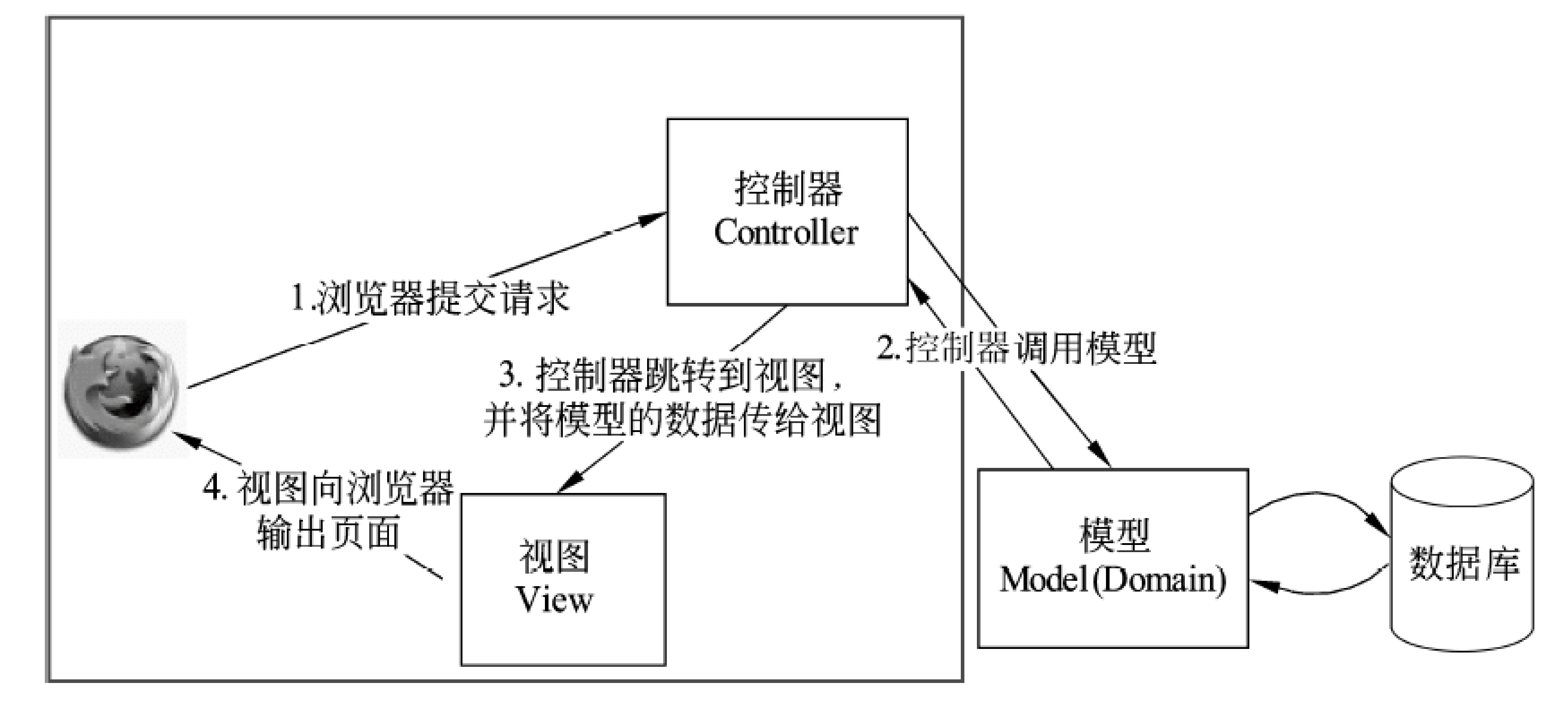


图 2-10　当前例子中仅包含控制器与视图

2.4　Scaffold 应用程序

Grails 中的一个 Domain 类，可以被简单地理解为是数据库中的一张表。Grails 推荐的开发方法与传统方法不同：不再是先设计项目的数据库，而是开发人员用 OO（Object Oriented，面向对象）的思想对数据进行 OO 建模（设计数据 Domain 类），然后运行 Grails，Grails 会根据 Domain 类的内容自动地生成数据库中的表。当然，这个自动创建数据库不一定是最优的，字段长度、索引等还需要由有经验的 DBA 进行调整优化。

Grails 在相当程度上弱化数据库在项目中的作用。当然，这种自动创建的数据库不可能满足所有的情况，至少对于历史遗留的数据库就是不适用的。本书第 11 章会对 GORM 的数据库映射做深入的讨论，这里先只讨论最简单的情况：单表自动映射。

Grails 中创建 Domain 的命令是 create-domain-class，如下所示：

```
>grails create-domain-class Student
Welcome to Grails 1.0.4 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: D:\grails-1.0.4

Base Directory: D:\workspace\HelloGrails
Note: No plugin scripts found
Running script D:\grails-1.0.4\scripts\CreateDomainClass.groovy
Environment set to development
     [copy] Copying 1 file to D:\workspace\HelloGrails\grails-app\domain
Created Domain Class for Student
     [copy] Copying 1 file to D:\workspace\HelloGrails\test\integration
Created Tests for Student
```

可以看到，Grails 创建了两个文件：一个是 Student 类；另一个是对应的测试类。使用一个文本编辑器打开 grails-app\domain\Student.groovy，修改其内容如下：

```
class Student {
    String name //姓名
    String sid  //学号
    String gender //性别
    String email
    Date enrollDate
    static constraints = {
        name(size:3..10)
        sid(matches:/\d{8}/)
        gender(inList:['男','女'])
        email(email:true)
        enrollDate()
    }
}
```

这里为 Student 类添加了几个属性，然后为这些属性添加了约束条件：

（1）name，长度在 3～10 之间；

（2）sid（学号），为 8 位数字；

（3）gender（性别），为“男”或“女”；

（4）email，必须为 E-mail 格式；

（5）enrollDate()，没有做约束条件。

然后再修改一下 HelloController 的内容如下（读者也可以自己再单独创建一个 Controller）：

```
class HelloController {
    def scaffold = Student
}
```

重新运行程序（grails run-app）[1]，可以看到一个针对 Student 表的 CRUD 程序。首先，能看到查看列表的页面，如图 2-11 所示。

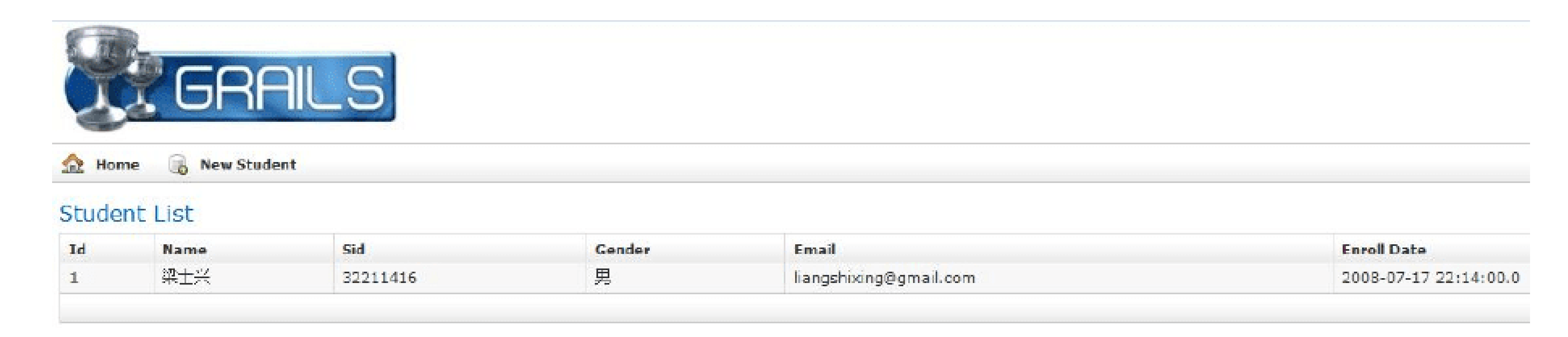


图 2-11　查看 Student 列表的页面

还可以单击链接看到查看单条记录（明细）的页面，如图 2-12 所示。

当然，也可以在页面上添加或编辑数据，如图 2-13 所示。

而且这个添加或编辑的页面是有数据验证功能的（不过，默认的错误信息还不够友

[1] 对于已经在运行状态的 Grails 程序，可以在控制台按 Ctrl+C 键来杀死进程。

好），如图 2-14 所示。

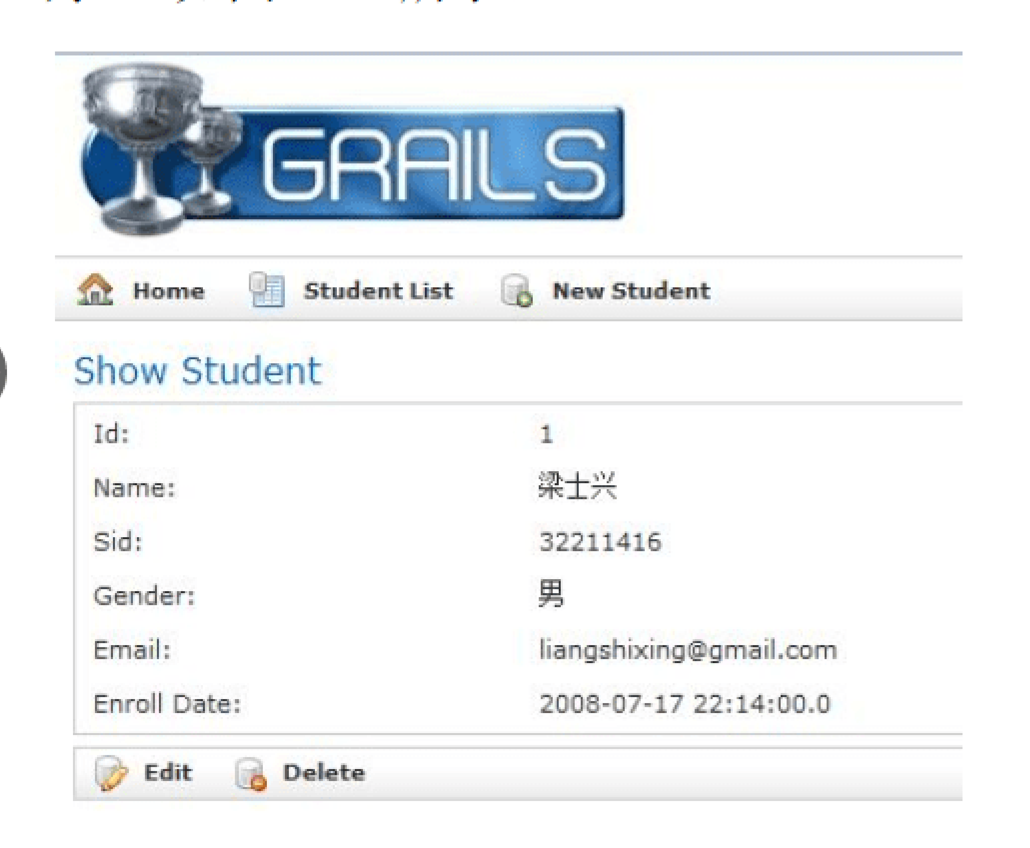


图 2-12　查看 Student 的页面

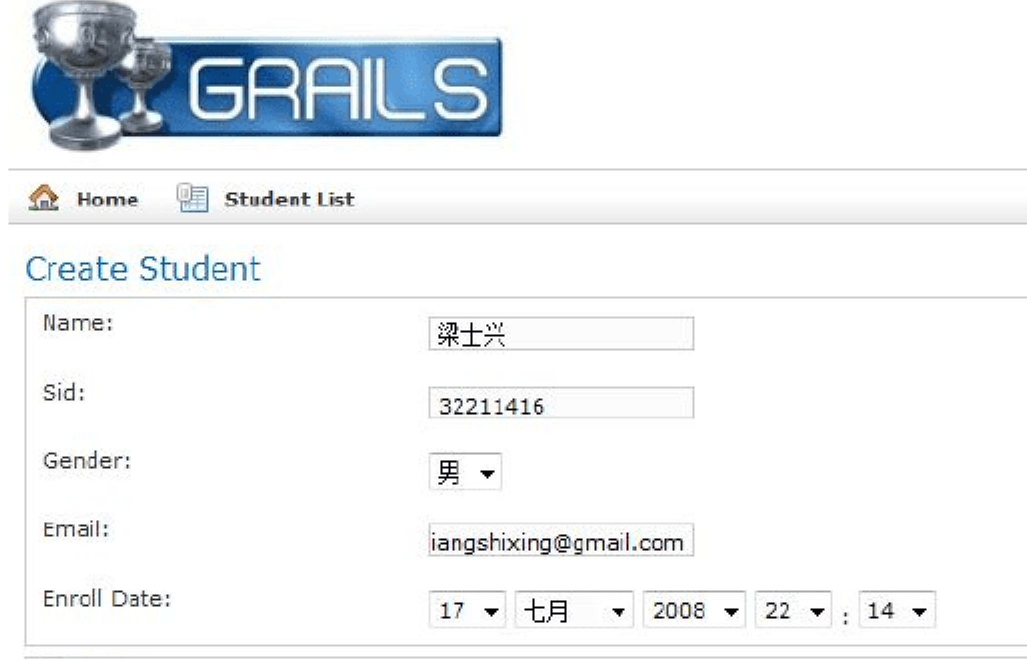


图 2-13　编辑 Student 的页面



图 2-14　编辑 Student 的页面（含错误信息）

这里只用了不到 20 行代码，花了不到 1 分钟，就可以完成这样一个“复杂”的任务。想象一下对于传统的 SSH（Struts+Spring+Hibernate）开发方式，即使是再熟练的程序员，也不可能在这么短的时间内完成这个任务。这就是 Grails 的魅力所在。当然，Scaffold 应用的实用性并不强，因为此时的页面是无法自由定制的，所以在实际的使用过程中，更多的是使用 Grails 的 generate*命令去生成 Controller 或 GSP，这些知识会在后面的章节中详细介绍。

细心的读者可能会发现一个问题，之前一直在强调 Domain 是针对数据库表的映射。那 Student 类映射的是数据库的哪张表？配置过数据库吗？程序已经运行起来了，那数据库在哪？

是的，这里并没有配置过数据库，但 Grails 默认捆绑了一个名叫 hsqldb 的数据库，这是一个非常轻量级的数据库。虽然实际项目中并不推荐用它，但用它快速实现一个能跑

起来的 Scaffold 应用，却是一个非常好的选择。关于数据库的配置，后面的章节将会做详细的介绍，到时候会把数据库换成 MySQL，并给出 MySQL 数据库在 Grails 中的配置方法。

2.5 开发工具的使用

读者可能会奇怪为什么没有先介绍开发工具的使用。这其实也体现了使用 Grails、RoR 开发的一个特点，那就是不强调工具的作用（这一点正好与 ASP.NET 完全相反）。但不管怎样有工具的帮助对提高开发效率，都是有好处的。

主流的 Java IDE，如 Eclipse、Netbeans、IntelliJ.IDEA 都可以找到针对 Grails 或 Groovy 的插件。其中对 Grails 支持比较好的是 Netbeans 6.5 和 IntelliJ.IDEA 7.0。由于 Netbeans 是免费的开源软件而 IntelliJ.IDEA 是收费的商业软件，因而强烈推荐使用 Netbeans 6.5 作为 Grails 的开发工具。

Netbeans 原本是最优秀的 Java IDE 之一，现在的 6.5 版，更是对 Grails 也提供了强大的支持。在 Netbean 的官方网站http://www.netbeans.org/downloads/index.html，可以下载到最新版 Netbeans，当前的最新版本是 6.5。

若已安装好 Netbeans，启动 Netbeans 后执行“工具”|“选项”命令，配置 Grails 的安装路径，如图 2-15 所示。

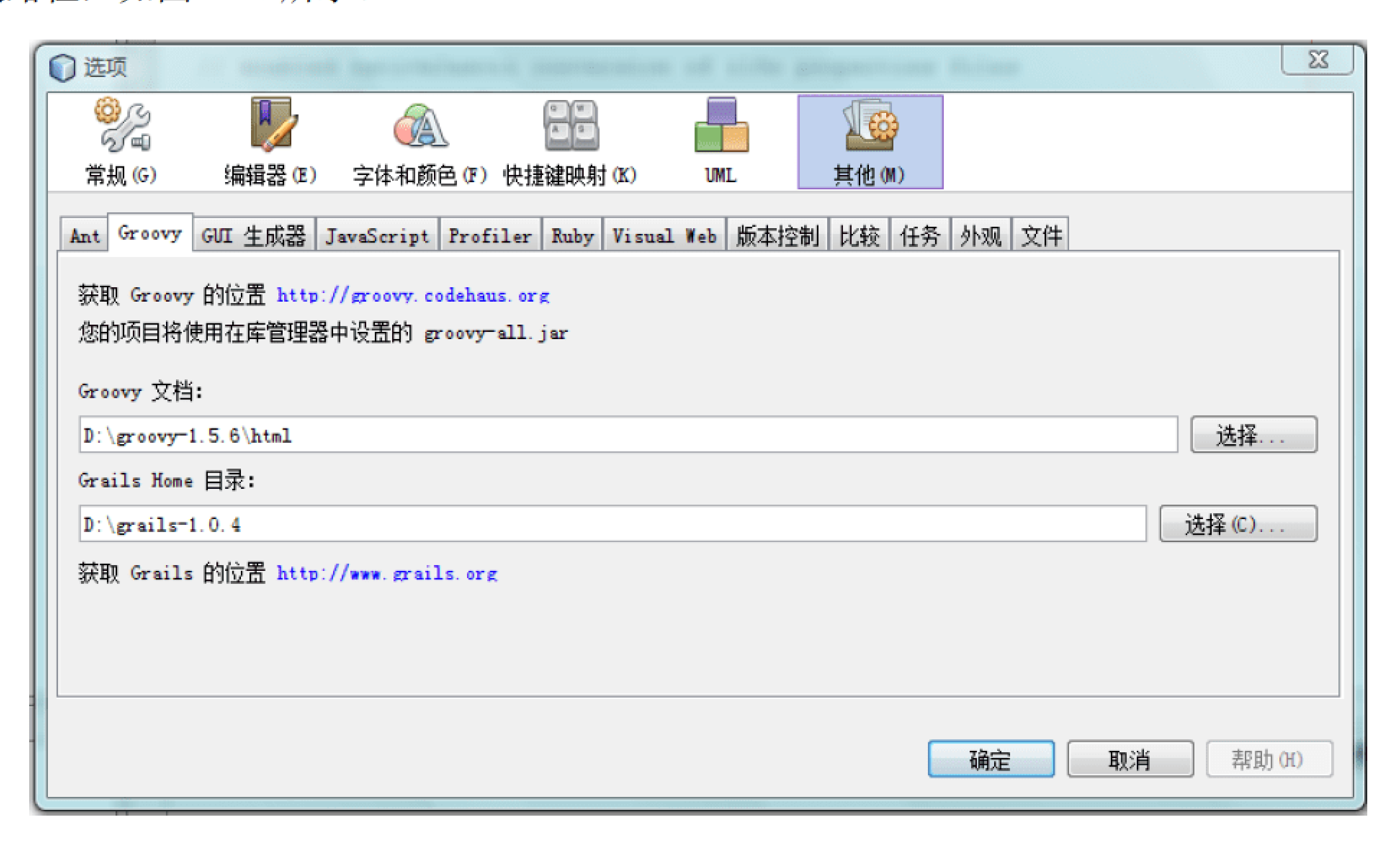


图 2-15 在 Netbeans 中配置 Grails

可以用 Netbeans 打开刚才创建的 Grails 工程，也可以在 Netbeans 中创建新的工程。Netbeans 对 Grails 支持有一个非常好的特性，它是非“侵入性”的。在 Netbeans 中新建或打开已有的 Grails 工程，都不会在硬盘上创建任何附加的文件。使用 Netbeans 开发 Grails 应用程序，会带来一些明显的好处。

1. 友好的代码编辑界面

Netbeans 对 Groovy 语言提供了较好的支持，包括代码高亮和代码提示。尽管对于脚本语言来说，实现 100%可靠的代码提示是非常困难的，Netbeans 的代码提示功能仍然可以对开发 Groovy 程序带来较大的帮助，如图 2-16 所示。

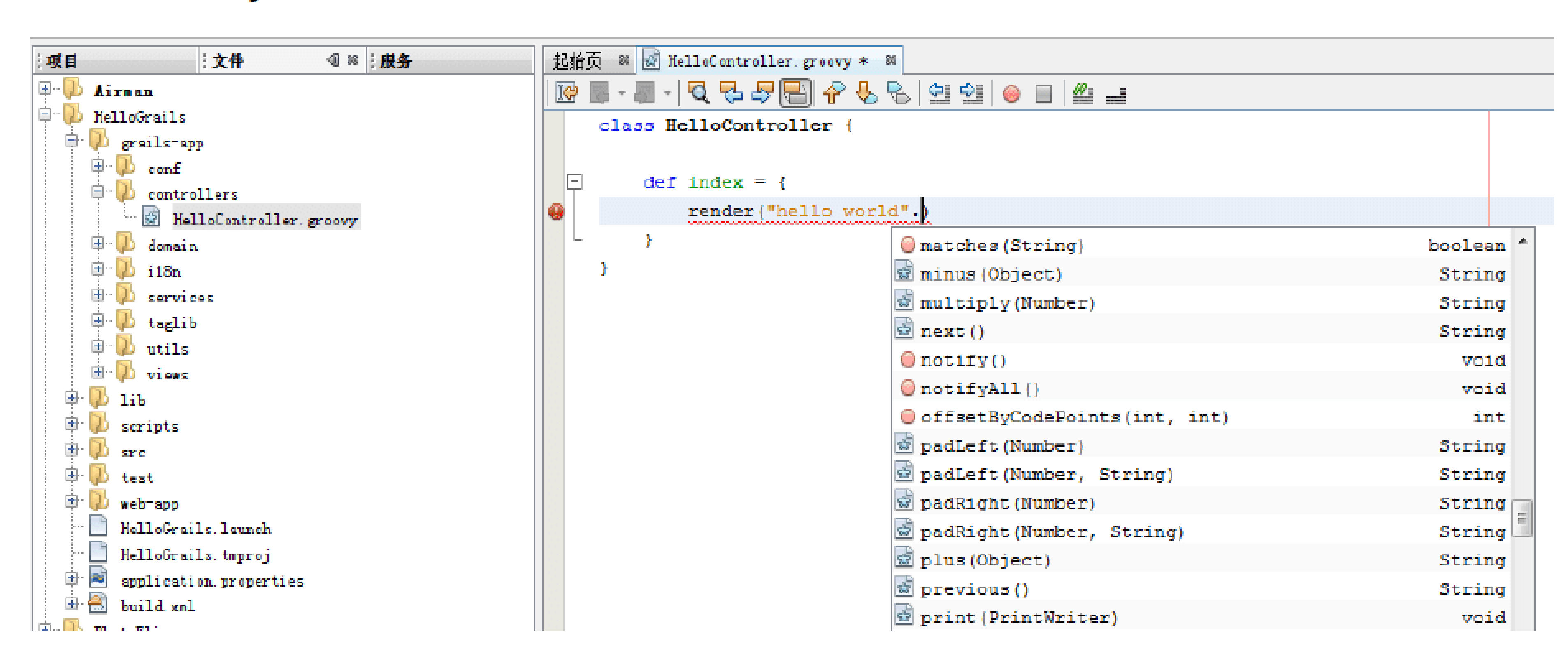


图 2-16　Netbeans 中 Groovy 的编辑效果

2. 简便的执行 Grails 命令

在前面的内容中介绍了一些 Grails 提供的命令，例如 create-controller、create-domain-class 等。很多用户都不习惯在控制台上输入命令，而更喜欢通过可视化的 UI 环境编辑内容。Netbeans 为 Grails 的大部分命令设计了菜单，用户可以通过单击菜单，执行相应的操作，如图 2-17 所示。

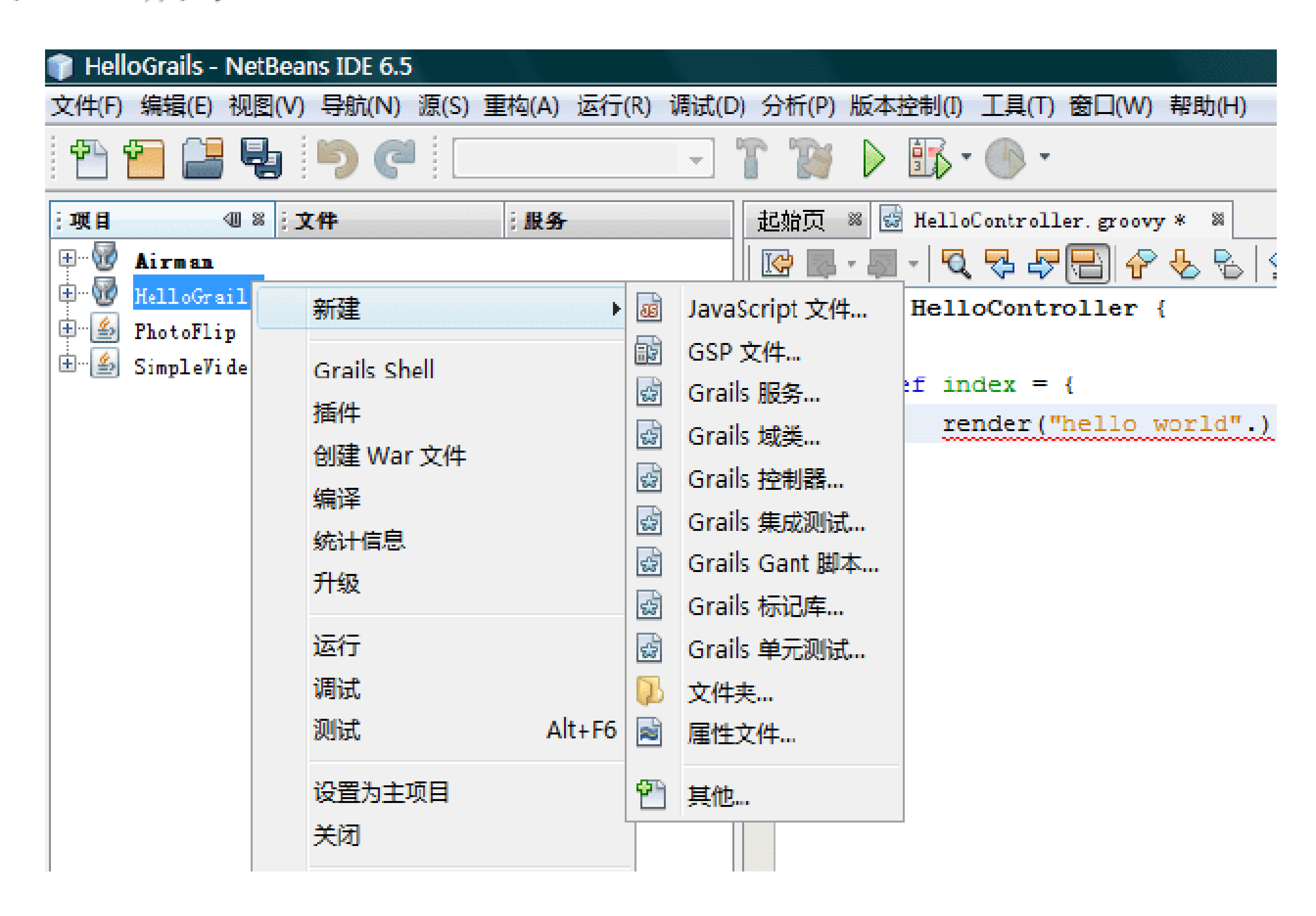


图 2-17　在 Netbeans 中执行 Grails 命令

3. 强大的 HTML、CSS 编辑功能

显然，对于 Web 应用的开发，IDE 的 HTML 编辑能力会极大地影响程序的开发效率。

Netbeans 早在 3.X 的时候就已经提供了良好的 HTML 编辑能力，此外，它还提供了一个可视化的 CSS 编辑器。这些功能对于开发 Grails 应用，虽然不是必需的，但无疑可以明显地改善开发效率和质量。

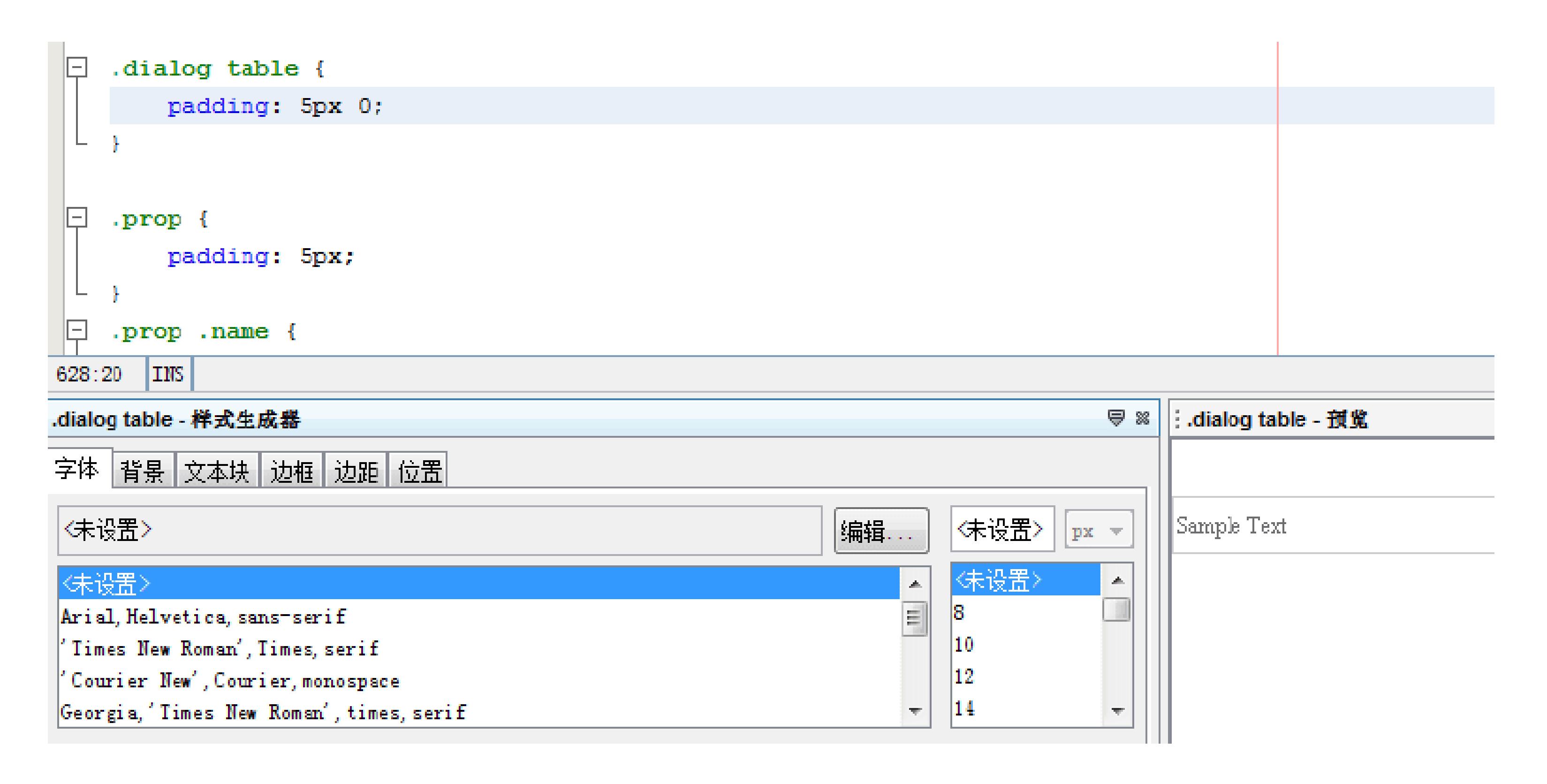


图 2-18　可视化的 CSS 编辑器

事实上，Netbeans 提供的功能远远不止上面介绍的这些。要了解更多 Netbeans 相关的知识，可以参考它的帮助文档。

2.6　本章小结

本章创建了第一个 Grails 程序。从 Grails 的安装开始，用 Grails 的 create-app 命令创建了一个 HelloGrails 项目。然后讨论了一个 Grails 的 MVC 结构，用 create-controller 命令创建了一个名为 hello 的 Controller，通过它学习了 Grails 的 URL 中 Controller 与 action 的组织方式。接着，用 create-domain-class 命令创建了一个 Domain 类，在 Controller 中用了一行代码“def scaffold = Student”就为它生成了可以进行增删查改的 CRUD 页面。本章的最后部分，简单介绍了如何使用 Netbeans 工具开发 Grails 应用，在 IDE 帮助下，Grails 应用的开发会变得更加容易。

第3章 Groovy VS Java

上一章，读者对 Grails 有了一点初步的认识，但对上一章中的代码，可能多少会有些不适应，毕竟 Groovy 不是 Java。引用 Groovy 官方的说法：它是运行在 JVM 之上的敏捷动态语言，它基于 Java 却为 Java 带来了大量强大的新特性，这些特性源于 Python、Ruby 和 Smalltalk。

Groovy 的灵活与简单，使得大量基于 Groovy 的 DSL（Domain Specified Language，某一领域的专用语言）被开发出来。使用 DSL，使得很多原本复杂的任务得到简化，例如，查询数据库、解析 XML、使用 Swing 开发桌面应用、配置 Spring 等。Grails 正是充分利用了 Groovy 的这些特性，开发了大量的 DSL，从而简化了 Web 应用的开发。

本章的目标就是通过使用尽可能少的文字，对 Groovy 语言与 Java 的区别做概要性的介绍，使得读者在阅读后面章节的时候不会有太多语法上的障碍。对于已经了解 Groovy 语法的读者，本章可以跳过不看，直接阅读后面的内容。

本章的例子程序不需要在 Grails 环境中运行，读者可以下载并安装独立的 Groovy 到本机，具体做法与 Grails 的安装相似：首先下载 Groovy 的安装包（可以在 http://groovy.codehaus.org 下载）；解压到本机的相关目录下，然后配置环境变量，如图 3-1、图 3-2 所示。



图 3-1　Groovy 的环境变量（1）



图 3-2　Groovy 的环境变量（2）

然后在控制台输入 groovy –v，如果看到如下提示，说明安装成功了。

```
>groovy -v
Groovy Version: 1.5.6 JVM: 10.0-b22
```

Groovy 给用户提供了一个小巧易用的工具，用于运行简单的 Groovy 代码。执行“开始”→“运行”命令，在弹出的对话框中输入 groovyConsole，会启动一个基于 Swing 的图形窗口，可以在里面编写简单的 Groovy 程序，如图 3-3 所示。

执行 Script|Run 命令，可以运行程序；执行 Script|Run Selection 命令可以只运行选中的代码，十分方便。接下来进入 Groovy 的正题。

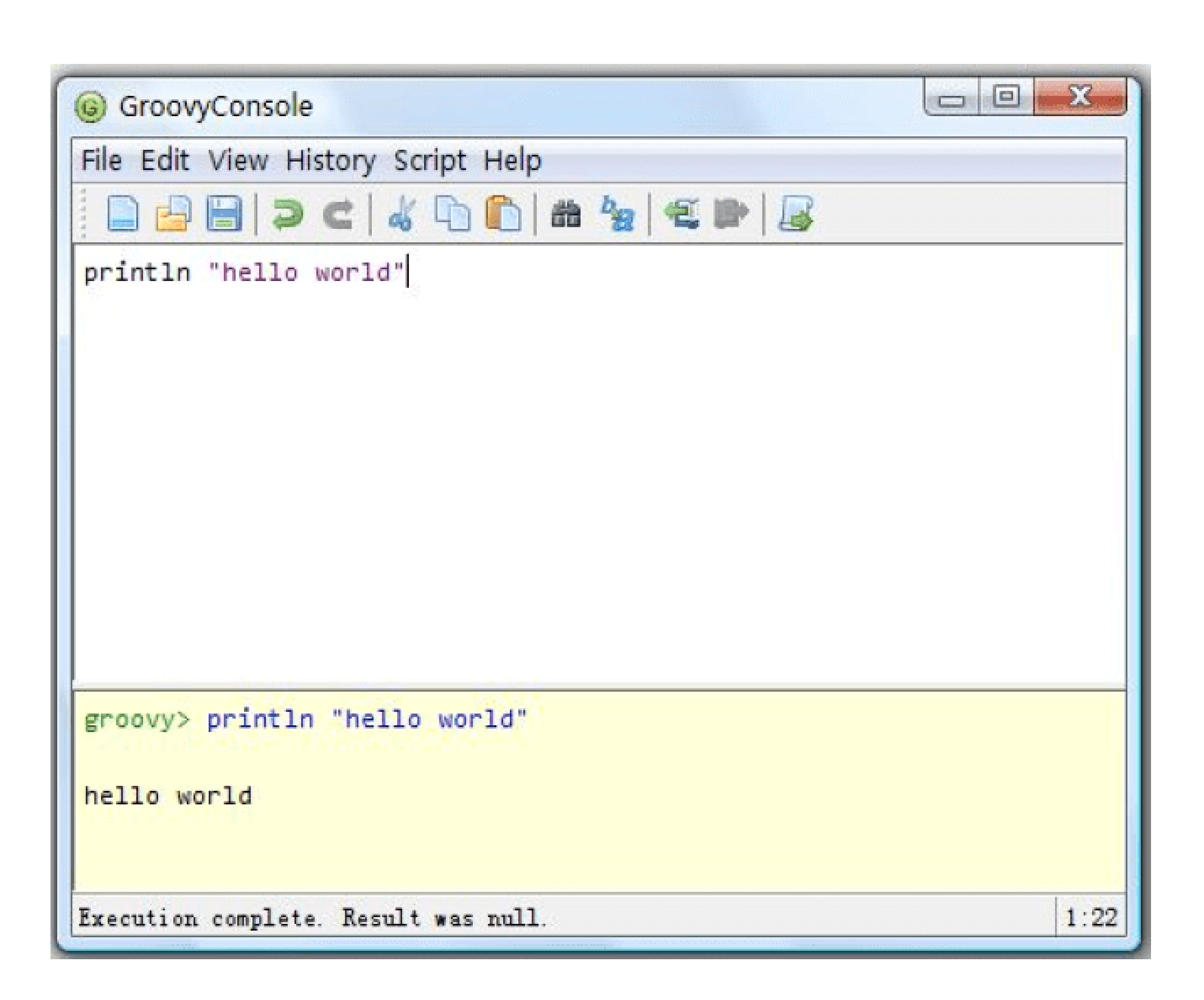


图 3-3　Groovy 控制台

3.1　Groovy 的基本类型与运算符

在 Groovy 中，一切都是对象。这点与 Java 不同，Java 中 int、double 等类型为基本类型，其变量不是对象；而在 Groovy 中，任何变量都是实实在在的对象。

运算符在 Groovy 中相当于是 Groovy 对象的方法，因而可以很容易地对运算符进行重载。

3.1.1　字符串

在 Groovy 中，字符串的功能变得更加强大，字符串的使用变得更加简单。

单引号“'”、双引号“"”、斜杠“/”都可以表示字符串。

单引号“'”表示静态的字符串，与 Java 中的字符串功能类似；双引号“"”表示的是动态的字符串，这是动态语言所独有的特性；“/”用于定义无转义符的字符串，常用于书写正则表达式，如下面的一些实例所示。

```
def str1 = 'hello'
def str2 = "$str1 world" //字符串中包含表达式的运算，所以是动态的
assert str2 == 'hello world'
def str3 = /\d\r\n\a\b/
assert str3 == '\\d\\r\\n\\a\\b'
```

这里的 def 是 Groovy 中定义变量的最常见的方法，它表示要定义一个变量，而不关心

这个变量是什么类型，变量的类型取决于给它赋值的类型。这种类型动态化的特点，是动态语言区别于静态语言的一个重要标志。assert 是 Groovy 的一个常用的方法：断言。含义是若断言的内容为假，则报异常退出程序。

如果连续使用 3 个引号（“"""” 或 “''”），则表示字符串段落：

```
def str4 = """
hello
world
hello
grails
"""
assert str4 == "\nhello\nworld\nhello\ngrails\n"
```

字符串常见的运算符操作：

```
assert str1 * 2 == 'hellohello' //字符串乘以数字
assert str2 - str1 == ' world' //字符串减法,会删除 str2 中左边第一个匹配的 str1
assert str2[0..3] == 'hell'
assert str2[1,3] == 'ell'
assert str2[-3..-1] == 'rld'
```

a..b 的结构在 Groovy 中叫做区间（Range），后面的小节会有介绍。

字符串常见的方法：

```
assert str2.toList() == ["h", "e", "l", "l", "o", " ", "w", "o", "r", "l",
                    "d"]
assert str1.reverse() == "olleh"
assert str2.size() == 20
```

3.1.2 数字

Groovy 中的数字是对象，因而可以在数字上调用方法：

```
def x =3
def y = 4
assert x + y == 7
assert x.plus(y) == 7
assert x instanceof Integer
def a = 2 / 3 // 0.6666666667
assert a instanceof BigDecimal
a = 2.intdiv(3)
assert a instanceof Integer
def b = a.setScale(3, BigDecimal.ROUND_HALF_UP)
assert b == 0.667
```

Groovy 的数学运算默认会有较高计算精度（因为使用了 BigDecimal 类型）：2/3 的结果不是 0 而是 0.6666666667。如果想要得到与 Java 程序类似的计算结果，需要使用 intdiv()方法。

3.1.3　Groovy 的类

Groovy 中类的定义与 Java 相比区别不大，额外注意以下两点即可。

一个特点是存在基于 Map 初始化的默认构造函数：Groovy 的类默认情况下（无需用户实现）可以使用一个 Map 作为构造函数的参数，对象的属性会被 Map 的同名 key 所对应的值初始化。

```
class A{
   String p1
   String p2
}
def a = new A(p1:"string1", p2:"string2")
assert a.p1 == "string1" && a.p2 == "string2"
```

另一个特点是属性字段。在 Groovy 中，没有定义访问限定符（如 public、private、protected）的字段，表示该字段是一个属性。这点不同于 Java。Java 类中默认的访问域是 package，但在 Groovy 中，默认是属性域。

```
class A{
   String p1
}
class A{
    private String p1
    public String getP1() { p1 }
    public void setP1(String p1) { this.p1 = p1 }
}
```

在 Groovy 的写法中，上述两个类是完全等价的。但明显前者的代码要少许多。同时，访问对象属性的过程也被简化了。调用 a.p1 和调用 a.getP1()是等价的，调用 a.p1=XX 与 a.setP1（XX）还是等价的。

Groovy 还给用户提供了一个非常实用的运算符："?"。例如，当要访问 a.b.c 的时候，难免要做一些判断以保证 a 和 b 都不是 null，否则一旦为空，将会抛出异常。写出来的代码就应该是这样：

```
if(a!=null) {
    if(a.b != null)
        return a.b.c
    else
        return null
```

```
} else {
    return null
}
```

这里有比较多的 if 和 else。Groovy 中的写法是：

```
a?.b?.c
```

怎么样？简单而又实用吧！

3.1.4 运算符

Groovy 中的运算符大多会对应为一个 Groovy 类的成员方法。例如，“+”运算符实际上是调用了对象的 plus 方法。表 3-1 列出了 Groovy 中常见的运算符和对应的类成员方法。

表 3-1 Groovy 基于方法的运算符

运算符	名字	方法	应用于
a + b	加	a.plus(b)	Number, string, collection
a | b	减	a.minus(b)	Number, string, collection
a * b	乘	a.multiply(b)	Number, string, collection
a / b	除	a.div(b)	Number
a % b	求模	a.mod(b)	integral number
a++	后自增	a.next()	Number, string, range
++a	前自增		
a--	后自减	a.previous()	Number, string, range
--a	前自减		
a**b	求幂	a.power(b)	Number
a | b	逻辑运算，“或”	a.or(b)	integral number
a & b	逻辑运算，“与”	a.and(b)	integral number
a ^ b	逻辑运算，“异或”	a.xor(b)	integral number
~a	逻辑运算，“非”	a.negate()	integral number, string（~运算符后面的返回一个正则表达式）
a[b]	下标	a.getAt(b)	Object, list, map, String Array
a[b] = c	下标赋值	a.putAt(b, c)	Object, list, map, StringBuffer, Array
a << b	左移位运算	a.leftShift(b)	integral number, 也用于 StringBuffers, Writers, Files, Sockets, Lists 的合并
a >> b	右移位运算	a.rightShift(b)	integral number
a >>> b	无符号的右移位运算	a.rightShiftUnsigned(b)	integral number

续表

运算符	名字	方法	应用于

switch(a){
case b:
}
	分类	b.isCase(a)	Object, range, list,
			collection, pattern, closure;
			也用于从集合 c 中查找所有符合条件 b 的 items，c.grep(b) b.isCase(item)

a == b	相等	a.equals(b)	Object
a != b	不等于	! a.equals(b)	Object
a <=> b	比较运算符，大于返回 1，小于返回 –1，等于返回 0	a.compareTo(b)	java.lang.Comparable
a > b	大于	a.compareTo(b) > 0	
a >= b	大于或等于	a.compareTo(b) >= 0	
a < b	小于	a.compareTo(b) < 0	
a <= b	小于或等于	a.compareTo(b) <= 0	
a as type	强制类型转换	a.asType(typeClass)	任意类型

如果想让自己定义的类能够支持运算符，只需要它提供了相应的方法即可，例如，定义复数类如下：

```
class Complex {
    int r
    int i
    def plus(other) {
        new Complex(r : this.r + other.r, i : this.i + other.i)
    }
}

def c1 = new Complex(r:10, i:10)
def c2 = new Complex(r:10, i: -5)
def c = c1 + c2
assert c.r == 20 && c.i == 5
```

3.2 Groovy 的控制结构

Java 的控制结构不外乎是：if 语句、switch 语句、for 循环、while 循环和 do 循环。Groovy 在这一点上，和 Java 是一样的。早期版本的 Groovy 不支持 Java 风格的 for 循环，但从 1.5 版开始就支持了。

Groovy 的 if 与 Java 有一定区别。Groovy 的 if 可接受的类型不仅仅是 boolean，还可以是数字、字符串、对象、集合。当传入 boolean 或 Boolean 时，处理逻辑与 Java 相同；当传入数字时，非 0 为真，0 为假；当传入字符串时，空字符串""为假，其他为真；当传

入集合时，元素个数等于 0 为假，元素个数大于 0 为真；当传入一个普通 object 时，null 为假，否则为真。下面是一些使用举例。

```
if(true && Boolean.TRUE )
    assert true
else
    assert false

if(1)
    assert true
else
    assert false

if("")
    assert false
else
    assert true

if([])
    assert false
else
    assert true

def a = null
if(a)
    assert false
else
    assert true
```

这个特性可以帮程序员减少很多没必要的代码，事实上这个特性是源于 Groovy 的类型转换。上面的几种类型在转换为 boolean 时，会使用上述的转换逻辑。换句话说，除了 if 语句外，Groovy 的 while 循环、do 循环等一切需要进行“真”、“假”判断的地方，都有上述的特性。

Groovy 的 for 循环支持两种风格：一种是传统 Java 风格；另一种是集合遍历风格（关于集合的知识下一节会有介绍）。其使用如下例所示：

```
//传统风格
for(int i=0;i<=10 ;i++)
    println i

//集合遍历风格
for(x in 0..10)
    println x

for(x in [1,3,5])
```

```
    println x
```

switch 语句的功能在 groovy 中得到了不小的增强，case 子句内可以进行多种复杂的判断：

```
switch (10) {
case 0 : assert false ; break
case 0..9 : assert false ; break
case [8,9,11] : assert false ; break
case Float : assert false ; break
case {it % 3 == 0} : assert false ; break
case ~/../ : assert true ; break
default : assert false
}
```

3.3 Groovy 的集合

Groovy 中有常用的 3 种集合类型，分别是列表 List、映射 Map 和区间 Range。

3.3.1 列表

Groovy 中列表的本质就是 java.util.List，但 Groovy 提供了大量能够简化其操作的特性。List 的初始化代码可以简化，只需要在[]中直接写入初始的数据即可：

```
def list1 = [1,2,3]
```

List 支持大量的运算符和方法：

```
list1 = list1 * 2
assert list1 == [1,2,3,1,2,3]
assert list1.unique() == [1,2,3]
list1 = [1,2,3,1,2,3]
assert list1.unque() == (list1 as Set) as List //将 List 先转换为 Set，
                                               //可消除重复值，
                                               //再转回 List，以进行比较
assert list1.size() == 6
assert list1.reverse() == [3,2,1,3,2,1]

assert [1,2,3,4] + [5] == [1,2,3,4,5]
assert [1,2,3,4] << 5 == [1,2,3,4,5]
assert [1,2,3,4] + 5 == [1,2,3,4,5]
assert [1,2,3,4,1] - [1] == [2,3,4]
assert [1,2,3,4,1] - 1 == [2,3,4]
```

```
assert [1,2,3,[4,5]].flatten() == [1,2,3,4,5]        //展开 List 中的 List

assert [1,2,3,4].max() == 4
assert [1,2,3,4].min() == 1
assert [1,2,3,4,1].count(1) == 2                     //统计 1 的个数
assert [1,2,3,4].sum() == 10                         //求和
assert [1,3,2,4].sort() == [1,2,3,4]

assert ! [1,2,3].disjoint([3,4,5,6])                 //存在交集
assert [1,2,3].disjoint([4,5,6])                     //不存在交集
assert [1,2,3,4].intersect([5,4,3,1]) == [4,3]       //求交集
```

List 有一个非常有用的运算符“*”，它的含义是依次对 List 每个元素调用“*”后面的方法：

```
def list2 = [[1,1],[2,2],[3,3]]
list2*.unique()
assert list2 == [[1],[2],[3]]
```

3.3.2 映射

Groovy 中的映射本质上就是 Java 的 java.util.Map，但 Groovy 中的 Map 在创建和使用时都更加简单。创建 Map，用 key:value 的形式成对出现，初始化时的 key 默认会被当作字符串处理，其使用如下例所示：

```
def map = [key1: "value1"]
assert map == ["key1": "value1"]
assert map.key1 == "value1" //用“.key”去访问 map 的成员
assert map["key1"] == "value1"//用“[key]”去访问 map 的成员
assert map.keySet() == ["key1"]
```

可能会有读者提出问题，如果 key1 是个变量会怎样？下面的代码可以解释这个问题：

```
def key= "key123"
def map1 = [key: "value"]
assert map1 == ["key":"value"]
assert map1.key == 'value'
assert map1[key] == null

def map2 = [(key): "value"]
assert map2 == ["key123":"value"]
```

相信从上面代码中，读者一定能找到答案：在初始化 Map 时，如果想用某一变量的值作为 key，需要用括号将该变量括起来。

为初始化过的 Map 添加和删除 key 是非常容易的，如下例所示：

```
map1["newKey"]="newValue"
assert map1 == ["key":"value" , "newKey":"newValue" ]
map1.remove("key")
assert map1.key == null
```

3.3.3 区间

区间是 Groovy 特有的一种数据结构。虽然它的名字起的很数学化，但它却是非常实用的。区间的定义非常简单，共有两种形式：一种为闭区间“起始..结束”；另一种为左闭右开区间“起始..<结束”。如下是一些实例代码：

```
def range1 = 1..5
def range2 = 1..<5
assert range1.contains(5)
assert !range2.contains(5)
assert range1.size() == 5
assert range2.size() == 4
```

事实上，区间使用的数据类型并不局限于数字、字符串、日期等一切实现了 next()和 previous()方法的类型（回顾一下介绍运算符的小节，这两个方法对应的是++和－－运算符），都可以定义为区间，如下例所示：

```
def today = new Date()
def yesterday = today-1
assert (yesterday..today).size() == 2

assert ('a'..'c').contains('b')
assert 'b' in ('a'..'c')
```

区间类型最常见的用法是用于循环和 switch 语句，如下例所示：

```
//区间用于循环
def log = ''
for( i in 1..5)
    log += i
assert log == '12345'

//区间用于switch
def score = 36
switch(score){
case 0..<60 :
    println '不及格'
    break
case 60..<70 :
    println '及格'
```

```
    break

case 70..<85 :
    println '良好'
    break
case 85..<100 :
    println '优秀'
    break
case 100:
    println '完美'
default: throw new IllegalArgumentException()
}
输出:  不及格
```

前置“*”运算符可以把区间展开，用于构造 List，例如：

```
assert [* range1] == [1,2,3,4,5]
```

3.4 Groovy 的闭包

3.4.1 闭包的定义

闭包（Closure）是 Java 所不具备的语法结构。闭包就是一个代码块，用“{ }”包起来。此时，程序代码也就成了数据，可以被一个变量所引用（与 C 语言的函数指针比较类似）。闭包的最典型的应用是实现回调函数（callback）。Groovy 的 API 大量使用闭包，以实现对外开放。闭包的创建过程很简单，例如：

```
{ 参数 ->
代码...
}
```

参考下面的例子代码，定义了 c1 和 c2 两个闭包，并对它们进行调用：

```
def c1 = { println it }
def c2 = { text -> println text }
c1.call("content1")      //用 call 方法调用闭包
c2("content2")           //直接调用闭包
```

“->”之前的部分为闭包的参数，如果有多个参数，之间可用逗号分割；“->”之后的部分为闭包内的程序代码。如果省略了“->”和它之前的部分，此时闭包中代码，可以用名为“it”的变量访问参数。

闭包的返回值和函数的返回值定义方式是一样的：如果有 return 语句，则返回值是

return 语句后面的内容；如果没有 return 语句，则闭包内的最后一行代码就是它的返回值。

3.4.2 闭包的代表

每个闭包都有一个代表（delegate）属性，指定了闭包的代理对象（闭包可以直接访问该对象的成员方法和属性），通过 delegate 可以访问代表对象。默认情况下，delegate 与 this 是相同的，但可以手动修改 delegate 的值，使闭包访问其他对象的方法和属性，例如：

```
class Foo{
    def method() { println 'call method'}
    def c1 = { Closure c->
        assert delegate == this  //闭包 c1 的 delegate 与 this 相同
        if(c) {
            println c.delegate.class

            //修改传入闭包的 delegate，使得传入的闭包可以访问 Foo 的成员
            c.delegate = this // L1
            c()
        }
    }
}

def foo = new Foo()
def c2 =  {
    println delegate.class //L2
    method() //L3
}

//c2() 此时不能调用 c2
//因为 L3 行引用的 method()不是全局方法，所以此时 c2 中调用 method()会报错
```

代码运行的输出结果为：

```
class Script63
class Foo
call method
```

因为闭包 c2 是全局闭包，所以它的 delegate 就是全局的 Script 类，于是第一行会输出 ScriptXX。当执行完 L1 后，会将 foo 对象设置成为 c2 的 delegate，于是此后执行 L2 时会输出 class Foo；同理，在设置了 c2 的 delegate 为 foo 后，c2 的内部就可以访问 foo 的成员方法 method()了，所以此时不会报错。

3.4.3 闭包在 GDK 中的使用

GDK（Groovy Development Kit）中有大量的 API 是针对 Closure 进行操作的。

表 3-2　常见的以闭包为参数的方法

方法名称	作用
each	可以进行遍历，每次取集合的一个元素传入的闭包
findAll	遍历集合，每次取集合的一个元素传入的闭包，将能使闭包返回真的元素组成一个新的集合
collect	遍历集合，每次取集合的一个元素传入的闭包，将闭包的返回值构造成一个新的集合
any	遍历集合，依次取集合的一个元素传入的闭包，若有一个元素能使闭包返回真，则立即返回真；若没有元素能使闭包返回真，则返回假
every	遍历集合，依次取集合的一个元素传入的闭包，若全部元素都能使闭包返回真，则返回真；若有一个元素能使闭包返回假，则返回假

下面对每种方法分别举例说明。

1. each 方法

```
//Range、List、Map 的 each 方法，可以进行遍历，每次取集合的一个元素传入的闭包
def log = ""
(1..10).each( {[1]
    log += it
})
assert log == "12345678910"

[key1: "value1", key2: "value2"].each({
entry ->
    println "$entry.key $entry.value"
})
```

输出：

```
key1 value1
key2 value2

[key1: "value1", key2: "value2"].each({
key, value ->
    println "$key $value"
})
```

输出：

```
key1 value1
key2 value2
```

[1] Groovy 中在调用带参数的方法的时候，可以不写()，如“(1..10).each { … }”。笔者不十分推荐这么做，毕竟多写一个括号费不了多大功夫，但对理解程序结构会更加容易。

2. finaAll 方法

```
//findAll 语句将调用闭包时返回真的元素组成一个新的 List
assert [1,3,5,7,9].findAll ({
    it > 4
} ) == [5,7,9]
```

3. collect 方法

```
//collect 方法将闭包应用于集合的每个元素后，将返回值构造成新的列表
assert [1,2,3,4,5].collect ( {
    it * it
})  == [1,4,9,16,25]
```

4. any 方法

```
// any 和 every 方法用于判断集合是否有符合条件的元素
//any: 当集合中至少存在一个元素符合条件时，返回真，否则为假
assert [1,3,5,7,9].any ({
    it * it == it + 6
})
```

5. every 方法

```
//every: 当集合所有元素都符合条件时，返回真，否则为假
assert ! [1,3,5,7,9].every ({
    it * it == it + 6
})
```

可应用于闭包的特殊运算符：“引用.&”可以把一个普通的 Groovy 方法转换为闭包。

```
def hello(word) {
    "hello $word"
}
def c = this.&hello
assert c("world") == "hello world"
```

3.5 本章小结

本章对 Groovy 的语法知识做了简单的介绍，分别介绍了 Groovy 的基本类型、集合类型、控制结构和闭包。本章的目的就是通过尽可能少的文字，帮助读者了解 Groovy，使读者朋友在有了一定的基础后，阅读本书后面章节的代码不再因语法问题而产生困惑。接下来将使用 Grails 开发一个实用的 Web 应用程序。

第二篇

实 际 应 用

这部分包含 7 章，这几章将一步步地完成一个购物车的应用。该部分包含：商品维护、商品搜索、用户的注册与登录、创建购物车、下订单、系统后台管理、页面布局的统计设计、自动化测试、应用的部署等。通过这个应用，读者可以学习到 Grails 多方面的知识，可以解决许多在使用 Grails 开发的过程中会遇到的问题。

第4章 商品维护

4.1 准备工作

回顾一下前面的章节，可以知道，创建一个 Grails 的应用其实并不复杂。购物车的应用很具有代表性，可以涉及到 Web 开发的大多数基础知识，因此这里选用购物车做为应用实例。考虑到准备用 Grails 开发的购物车，就叫它 GDepot。好了，让代码运行起来再说。执行命令：

```
>grails create-app GDepot
```

Grails 帮用户创建好了应用程序框架，并且组织好了目录结构。这一次不再使用 Grails 提供的 hsqldb 数据库，换用高性能的开源数据库 MySQL，因而需要调整数据源的配置。

由于 Groovy 程序本质上也是 Java 程序，因此也需要使用 JDBC 驱动进行数据库的连接。首先把 MySQL 的 JDBC 驱动的 jar 包（可以从 MySQL 网站上下载到）mysql-connector-java-5.1.6-bin.jar 复制到项目文件夹的 lib 目录中去。然后修改数据源的配置，打开 grails-app\conf\DataSource.groovy。

```
dataSource {
   pooled = true
   driverClassName = "com.mysql.jdbc.Driver"
   username = "root"
   password = "mysql"
}
hibernate {
    cache.use second level cache=true
    cache.use_query_cache=true
    cache.provider_class=
        'com.opensymphony.oscache.hibernate.OSCacheProvider'
}
// environment specific settings
environments {
   development {
      dataSource {
         dbCreate = "update" // one of 'create', 'create-drop','update'
```

```
            url = "jdbc:mysql://localhost:3306/GDepot_dev"
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            url = "jdbc:mysql://localhost:3306/GDepot_test"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
            url = "jdbc:mysql://localhost:3306/GDepot_prod"
        }
    }
}
```

代码中粗体+斜体的部分就是需要修改的内容。dataSource 节点用于配置数据源，如果将数据源改换为 MySQL，就要修改其中的 driverClassName 以及用户名和密码。在 hibernate 节点中可以配置缓存相关的内容，这一部分会在后面有详细介绍，这里先使用默认值。environments 中包含了 3 个结构相同的子内容，分别对应开发环境、测试环境和产品环境。每一部分又分别对应数据库的创建策略（没错，数据库是程序自动创建的）和连接数据库的 URL。Grails 的 3 种数据库创建策略 dbCreate 分别表示如下含义：create-drop，当 Grails 运行时，如果目标数据库已存在，则 drop 删掉它并重新创建；create，当 Grails 运行时，如果数据库不存在，创建数据库，但如果目标数据库已存在，原有数据库的结构不会进行修改，会删除已有数据；update，如果目标数据库不存在，则创建数据库，如果数据库已存在，则根据需要（通常是 Domain 类的内容变了），自动更新数据库。当然，还可以选择不让 Grails 自动创建数据库，只要删除这个 dbCreate 选项就可以了[1]。

传统项目开发过程有对数据库进行 ER 建模的步骤，在 Grails 中已经不再存在。取而代之的是针对 Domain 的 OO 建模。程序员按照 OO 的思想和方式设计好 Domain 类，然后由 Grails 创建基于该 Domain 类的数据库表。当然，Grails 自动生成的数据库不一定是最优的，还需要 DBA 进行相应的优化。

这里首先创建商品和分类的 Domain 类：

```
>grails create-domain-class Goods
>grails create-domain-class Category
```

可以在 grails-app\domain 目录下看到 Category.groovy 和 Goods.groovy 两个文件，分别修改它们的内容如下：

```
class Goods {
```

[1] 这里强烈建议不要在 product 环境中使用 dbCreate 选项，产品环境的数据库应该是由 DBA 创建的经过优化的数据库。

```
    String title
    String description
    BigDecimal price
    String photoUrl
    Category category
}

class Category {
    String categoryName
    static hasMany = [goods:Goods]
    static constraints = {
        categoryName(unique:true)
    }
}
```

这里指定了商品和分类的属性（字段），同时指定了商品和分类的关系是一对多的关系。Category 类中的 constraints 闭包，定义了对 Category 表中数据的约束条件，即为 categoryName 字段添加唯一性约束[1]。接下来就运行程序，看看它自动生成的表是什么样子。运行命令：

```
>grails run-app
```

很不幸，可能会看到这样的报错：

```
...
Caused by: com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException:
Unknown database 'gdepot_dev'
       at com.mysql.jdbc.Util.handleNewInstance(Util.java:406)
       at com.mysql.jdbc.Util.getInstance(Util.java:381)
       at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:1030)
       at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:956)
       at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3491)
       at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3423)
       at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:910)
       at com.mysql.jdbc.MysqlIO.secureAuth411(MysqlIO.java:3923)
...
```

其实道理很简单，在配置 DataSource.groovy 的时候，指定的 URL 中包含了数据库的名称。但是，显然在尚未创建该数据库时，使用这样的 URL 不可能成功连接数据库。因而，必须先手动创建这 3 个环境的数据库：

```
>mysql -uroot -pmysql
mysql>CREATE DATABASE GDepot_dev;
mysql>CREATE DATABASE GDepot_test;
```

[1] 要了解更多数据约束与验证的知识，详见第 6.1 节表单验证与资源文件。

```
mysql>CREATE DATABASE GDepot_prod;
mysql>\q
```

创建好数据库后（其实现在只创建一个 dev 环境就够了），再次运行程序：

```
>grails run-app
```

这一次，程序可以正常地运行起来了。从数据库中，可以看到 Grails 创建了两张表，并为它们建立了外键：

```
CREATE TABLE 'goods' (
  'id' bigint(20) NOT NULL auto increment,
  'version' bigint(20) NOT NULL,
  'category_id' bigint(20) NOT NULL,
  'description' varchar(255) NOT NULL,
  'photo_url' varchar(255) NOT NULL,
  'price' decimal(19,2) NOT NULL,
  'title' varchar(255) NOT NULL,
  PRIMARY KEY  ('id'),
  KEY 'FK5DF97566285CD1A' ('category_id'),
  CONSTRAINT 'FK5DF97566285CD1A' FOREIGN KEY ('category_id') REFERENCES
  'category' ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE 'category' (
  'id' bigint(20) NOT NULL auto_increment,
  'version' bigint(20) NOT NULL,
  'category name' varchar(255) NOT NULL,
  PRIMARY KEY  ('id'),
  UNIQUE KEY 'category_name' ('category_name')
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
```

Grails 为每张表自动创建了一个名叫 id 的主键，和一个 version 字段（源于 Hibernate 的乐观锁（optimistic locking）机制）。并且 goods 表中的 category_id 字段实现了与 category 表的关联。

修改一下 grails\config\BootStrap.groovy 文件的内容，让程序启动时自动添加几条数据，修改情况如下所示：

```
class BootStrap {
    def init = { servletContext ->
            def category = new Category(categoryName:'Book')
            if(category.save()){
                println 'new category saved!'
                def allgoods = [ new Goods(title:'Grails',price:20.0,
                    description:'Grails Book...', photoUrl: ''),
                new Goods(title:'Groovy',price:20.0,
```

```
                    description:'Groovy Book.',photoUrl: '')]
                allgoods*.category = category
                allgoods*.save()
                println 'all goods saved!'
            }
        }
        def destroy = {
        }
}
```

这样在程序启动时，Grails 会自动添加一条商品分类和两件商品。

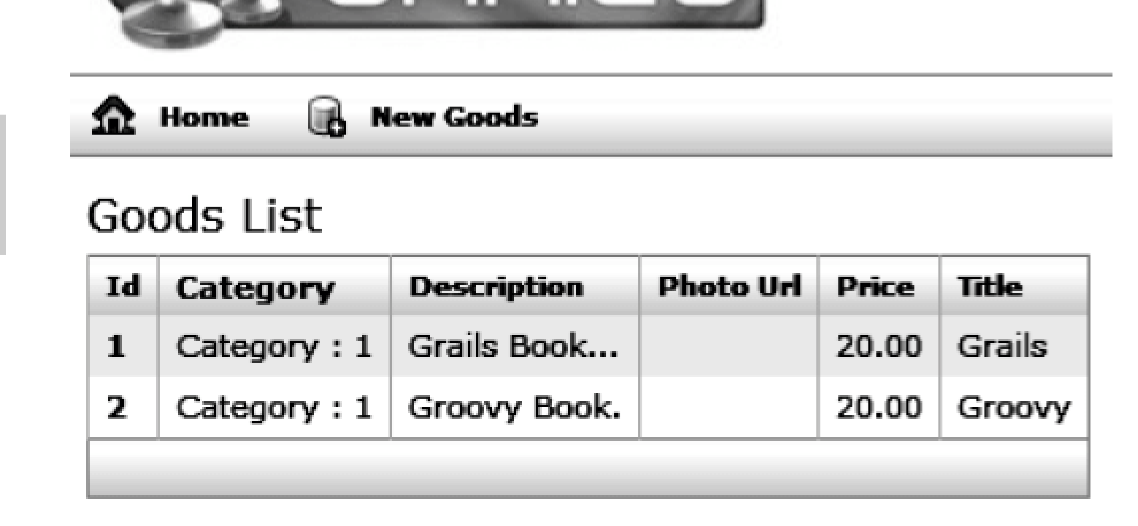
4.2 查看商品列表

完成了准备工作，这里有了数据库和少量初始数据。这一节将介绍：如何在页面上输出数据、如何用标签去遍历一个列表、如何去构造超级链接。

首先，让 Grails 生成这两个 Domain 的 CRUD 页面，命令如下：

```
>grails generate-all Goods
>grails generate-all Category
```

重新运行程序，可以看到 Grails 自动创建了两个骨架程序，可以通过浏览器访问 http://localhost:8080/GDepot/goods/list，显示效果如图 4-1、图 4-2、图 4-3 所示。



Id	Category	Description	Photo Url	Price	Title
1	Category : 1	Grails Book...		20.00	Grails
2	Category : 1	Groovy Book.		20.00	Groovy

图 4-1 商品列表



图 4-2 创建商品



图 4-3 商品明细

虽然客户不可能对这样的页面满意，但至少最基本的 CRUD 功能已经实现了。那么接下来，就着手去改进它。首先，要修改显示商品列表的页面（grails\view\goods\list.gsp），如下所示：

```
<table>
<tbody>
    <g:each in="${goodsInstanceList}" status="i" var="goodsInstance">
    <tr class="${(i % 2) == 0 ? 'odd' : 'even'}">
    <td><img src="${fieldValue(bean:goodsInstance, field:'photoUrl')}"
style="width:200px"/>
</td>
    <td style="width:500px">
        <g:link action="show" id="${goodsInstance.id}">
        ${fieldValue(bean:goodsInstance, field:'title')}
    </g:link><br/>
        Category: ${goodsInstance.category?.categoryName}<br/>
        ${fieldValue(bean:goodsInstance, field:'description')}<br/>
        Price: ${fieldValue(bean:goodsInstance, field:'price')}
        </td>
    </tr>
    </g:each>
    </tbody>
</table>
```

修改一下商品列表的显示方式，然后用它的编辑界面修改两条商品的记录，使它的 pictureUrl 不再为空，于是，再看商品列表的界面，是不是就舒服些了，如图 4-4 所示。

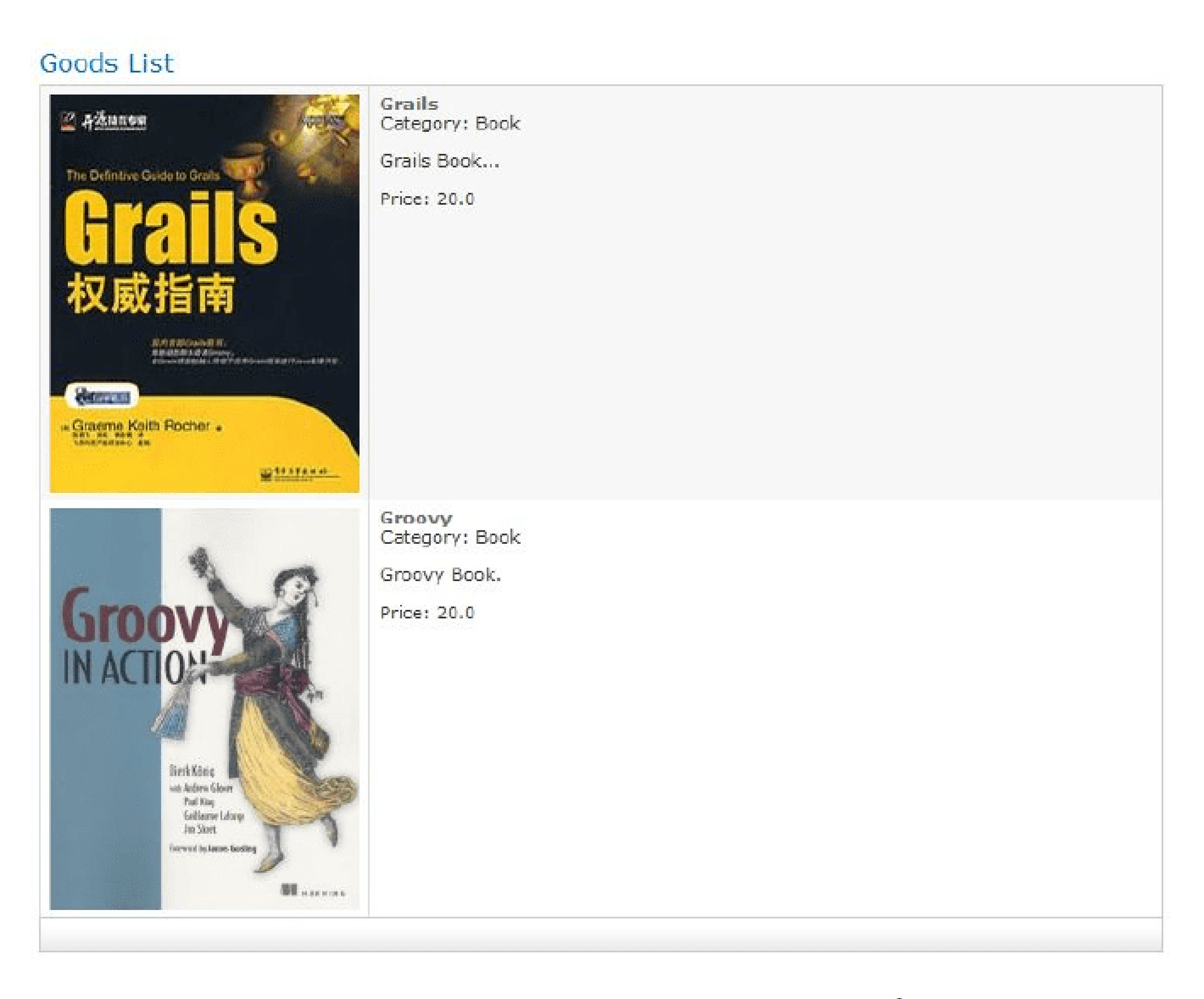


图 4-4　经过调整后的商品列表页面[1]

[1] 两个商品的图片的 pictureUrl 分别是 http://otho.douban.com/lpic/s3032092.jpg 和 http://otho.douban.com/lpic/s3369439.jpg。

这个页面功能虽然简单，还是需要细细品味一下。前面的第 2.3 节介绍了 Grails 页面的请求过程，首先是执行 Controller，Controller 可以接收用户提交的表单，可以为页面显示准备数据。因此，要了解 list 这个页的执行过程，就要先看 GoodsController 类的 list action：

```
def list = {
    if(!params.max) params.max = 10
      [goodsInstanceList: Goods.list( params ) ]
}
```

这里调用了 Goods 类的 list 方法，实现对数据库 Goods 表的查询。

Goods 类的代码是程序员亲手编写的，程序员从来没给它添加过什么 list 方法，但这里确实是调用了 list 方法实现对数据库的查询。动态语言可以在运行时改变行为，使已有的类增加新的方法或属性，Groovy 语言可以通过 metaClass 实现上述功能。这里不对 Domain 查询数据库的原理进行过多的阐述，在最后的第四部分将进行深入的讨论。

GSP 页面上可以访问 Controller 中 action 所返回值（Map）中的数据，即 goodsInstanceList。并且在 GSP 页面中使用一个<g:each>标签来对这个 List 的内容进行遍历，例如：

```
<g:each in="${goodsInstanceList}" status="i" var="goodsInstance">
```

g:each 标签包含 3 个属性：in 为必选，指定表示需要遍历的集合；status 为可选，返回当前对应的索引（行号）；var 为可选，用于指定每次取出元素的名称，如果不指定 var 属性，则自动指派为 it。通过 g:each 标签，实现了把数据库的每条记录显示为表格的一行。

${}可以对“{}”内部的表达式进行计算，然后输出其计算结果。例如：

```
${goodsInstance.category?.categoryName}
```

这将在页面上输出 goodsInstance.category?.categoryName 的值。

${fieldValue(bean:goods, field:'title')}，会输出 fieldValue 方法的执行结果。它的作用是取出某一个 bean 的某一个 field 的值（这个方法比较智能，能够判断该 field 的取值是否合法，还能自动进行 encodeAsHtml()操作，以防止跨站脚本攻击，因此推荐使用）。

g:link 是用于输出超级链接的标签，这也是 Grails 中最常见的一个标签。可以通过指定 controller 和 action 属性构造链接，也可以通过 id 或 params 属性传递参数，如果要构造传统的 URL，则可以使用 url 属性去直接构造，例如：

例 1：

```
<g:link controller="goods" action="show" id="${1}">Grails</g:link>
```

若不指定 controller，则使用当前的 controller。输出为：

```
<a href="/GDepot/goods/show/1">Grails</g:link>
```

例 2：

```
<g:link action="show" params="[id:1,style:'gray']">查看</g:link>
```

输出为：

```
<a href="/GDepot/goods/show?id=1&style=gray">查看</a>
```

例 3：

```
<g:link url="http://www.buaa.edu.cn">北京航空航天大学</g:link>
```

输出为：

```
<a herf="http://www.buaa.edu.cn">北京航空航天大学</a>
```

修改了商品列表的显示效果，还需要再修改显示商品明细的页面，只需要简单地修改一下显示格式即可。将 grails-app\views\goods\show.gsp 文件中的 tbody 标签的内容修改为：

```
<tbody>
    <tr class="prop">
    <td valign="top" class="name">Title:</td>
        <td valign="top" class="value">
          ${fieldValue(bean:goodsInstance, field:'title')}
        </td>
    </tr>
    <tr class="prop">
        <td valign="top" class="name">Photo:</td>
        <td valign="top" class="value">
        <img src="${fieldValue(bean: goodsInstance, field:'photoUrl')}"
            style="width:200px"/></td>
    </tr>
   <tr class="prop">
       <td valign="top" class="name">Category:</td>
       <td valign="top" class="value">
       <g:link controller="category" action="show"
          id="${ goodsInstance?.category?.id}">
          ${ goodsInstance?.category?.categoryName.encodeAsHTML()}
       </g:link>
       </td>
   </tr>
   <tr class="prop">
       <td valign="top" class="name">Description:</td>
        <td valign="top" class="value">
          ${fieldValue(bean: goodsInstance, field:'description')}
       </td>
   </tr>
   <tr class="prop">
       <td valign="top" class="name">Price:</td>
       <td valign="top" class="value">
          ${fieldValue(bean: goodsInstance, field:'price')}
       </td>
   </tr>
</tbody>
```

显示效果如图 4-5 所示。

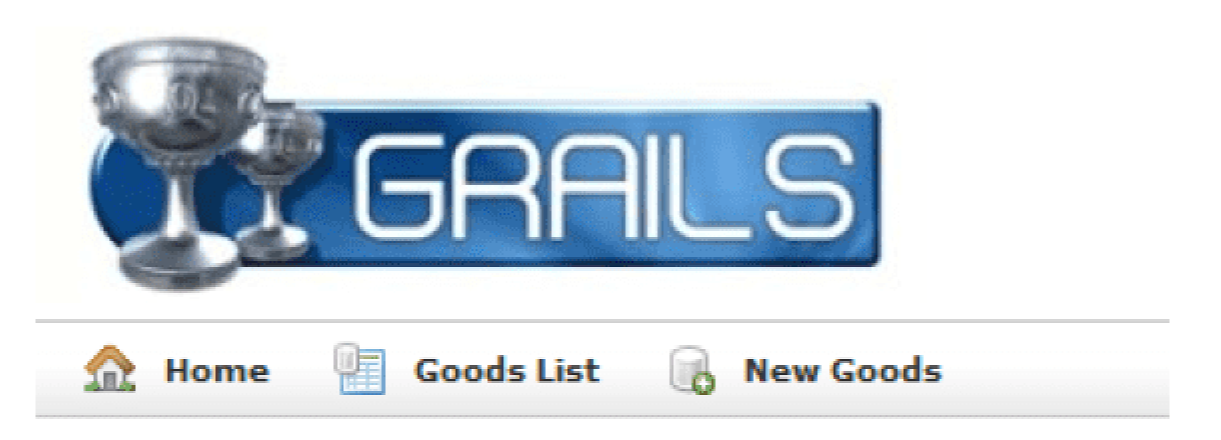




图 4-5　改进后的商品明细页面

4.3　创建和编辑商品

这里暂时完成了商品列表页面，下面去处理商品提交的表单页面。这一节包含更多的知识点，主要有：表单的构造、表单的接收、表单的输出。

首先打开编辑商品的页面[1]grails-app\views\goods\edit.gsp，需要先调整一下几个字段的顺序（category、title、description、photoUrl、price），然后把 description 的文本框替换为多行的。显示效果如图 4-6 所示。

这里仅摘录表单相关的内容（完整的代码修改情况，读者可以参阅本书附带光盘的代码 ..\grails-app\views\goods\edit.gsp）：

[1] 创建商品和编辑商品在逻辑上比较相似，因而这里仅以编辑商品为例。

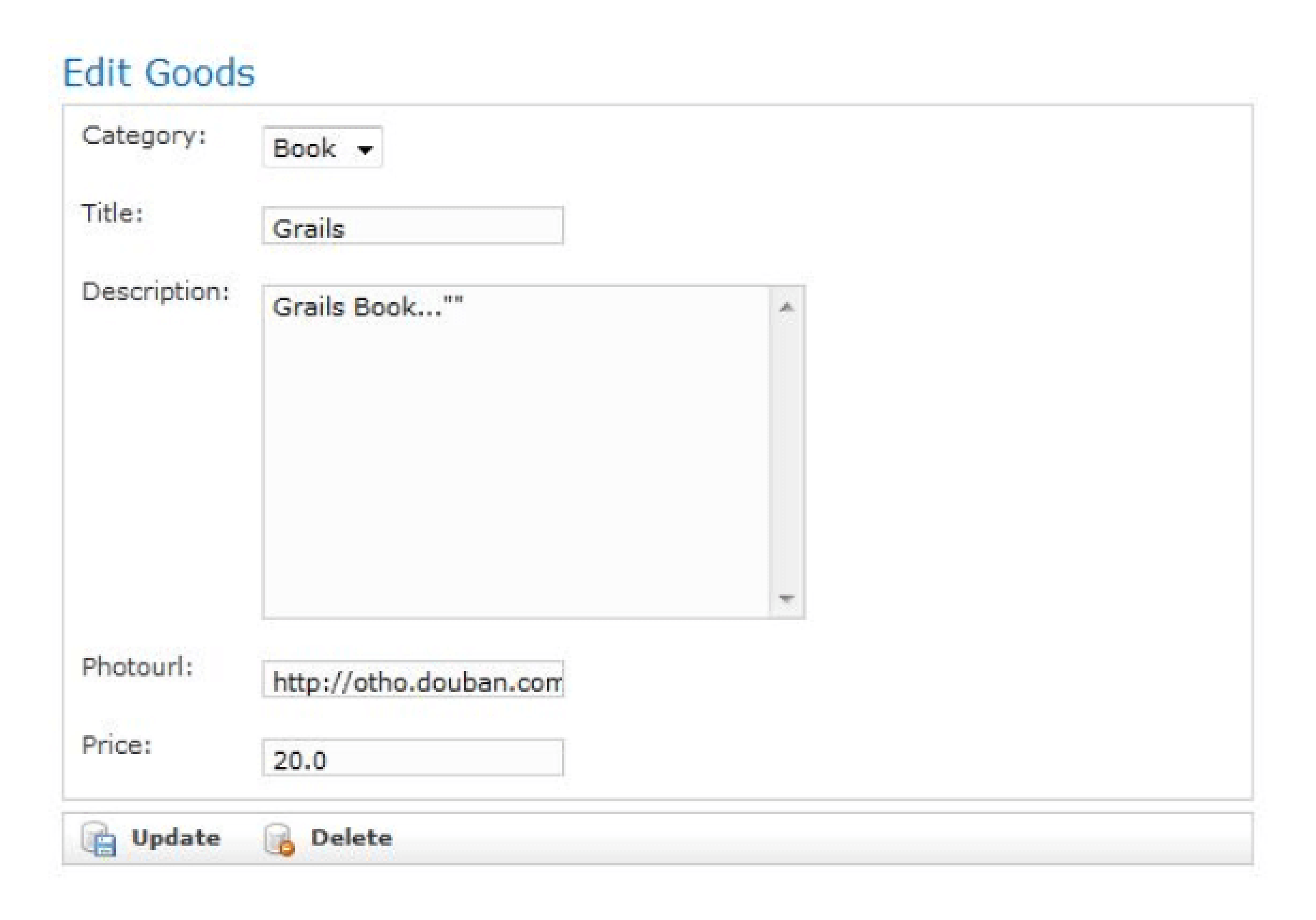


图 4-6　经过调整后的编辑商品页面

```
<g:form method="post" >
    <input type="hidden" name="id" value="${goodsInstance?.id}" />
<g:select optionKey="id" optionValue="categoryName"
 from="${Category.list()}" name="category.id"
 value="${goodsInstance?.category?.id}" >
</g:select>
<input type="text" id="title" name="title"
        value="${fieldValue(bean:goodsInstance,field:'title')}"/>
<textarea id="description"  name="description">${
        fieldValue(bean:goodsInstance,  field:'description')
}</textarea>
<input type="text" id="photoUrl" name="photoUrl"
 value="${fieldValue(bean:goodsInstance,field:'photoUrl')}"/>
   <input type="text" id="price" name="price"
     value="${fieldValue(bean:goodsInstance,field:'price')}" />
   <g:actionSubmit class="save" value="Update" />
<g:actionSubmit class="delete"
 onclick="return confirm('Are you sure?');" value="Delete" />
</g:form>
```

从上面的代码可以看出，Grails 的表单同标准的 HTML 表单几乎是完全一样的，换句话说，读者几乎不需要去学习和记忆 Grails 自带的表单标签，只要使用标准的 HTML 标签就够了（对比 Struts 或 JSF，这是很大的优势，毕竟学习成本降低了）。从上面的代码还能看到表单输出相关的知识点：标准的 HTML 表单项，只需要修改 value 属性值（这里有个例外是 textarea，它是把默认值写到<textarea></textarea>中间去）；而对于相对复杂的下拉列表框<select>，Grails 提供的 g:select 标签，也可以使用 value 属性去给它指定默认选中的项。

<g:form>标签是本节的第一个知识点。它用于输出一个传统的<form>标签。g:form 的

使用和 g:link 类似，支持用 controller、action、url 属性去构造表单接收 URL 的地址，如下面的一些示例。

例 4：

```
<g:form controller="goods" action="save">
```

输出：

```
<form action="/GDepot/goods/save">
```

例 5：

```
<g:form url="www.buaa.edu.cn">
```

输出：

```
<form action="www.buaa.edu.cn">
```

还可以指定表单提交的 method 及 enctype，如：

```
<g:form action="upload" method="post" enctype="multipart/form-data">
```

g:form 的使用和标准的 HTML 的 form 非常相似。使用它的主要好处是它可以使用 controller 和 action 去生成 url。不过这里要强调一点，仅通过生成 form 的 action 不能唯一决定最终的表单处理程序。**g:actionSubmit 标签也可以影响最终由哪个 controller 的哪个 action 处理表单**。

<g:select>是第二个知识点，这是一个非常有价值的标签，因为使用它要比使用传统的 HTML 的 select 要简便得多。g:select 有一个必填的 from 属性，这个属性用于指定其选项的数据来源。以上例为例，from="${Category.list()}"表示用 Category.list()的返回结果作为数据源，即下拉列表框的选项是 Category 的列表；optionKey="id"表示依次用数据库源中每个 Category 的 id 作为每个选项的值(即<option **value**=""> </option>中 value 对应的内容)；类似地，opitonValue="categoryName"表示依次用数据库源中每个 Category 的 categoryName 作为每个选项的显示结果（例如<option >**Book**</option>中 Book 对应的内容）。

value 属性是整个 g:select 最有价值的体现，通过指定 value 属性，g:select 会自动选中当前列表中与 value 等值的选项。倘若是自己来手动实现这个功能，恐怕就要在遍历选项的时候逐一比较了，这将变得苦不堪言。

g:actionSubmit 标签可以输出一个<input type="submit"/>。它有一个很强大的功能，就是指定响应按钮的 action，相当于给不同的按钮指定不同的事件处理程序。分析 GSP 输出的 HTML 代码：

```
<g:actionSubmit class="save" value="Update" /> 输出:
<input type="submit" name="_action_Update" value="Update" class="save" />
```

从中可以看出，提交表单中包含了 name 为_action_XXX 的 input 项(HTML 处理 submit 类型的按钮的方式是，用户单击了哪个按钮，就会提交它的 name 和 value)，Grails 会根据这个_action_XXX 后面的 XXX 去选择相应的 action（XXX 是首字母大写的 action 的名字，即，要运行 update，则应写成_action_Update）。注意一点，这里之所以说的是表单项而不

是按钮，是因为 Web 框架在 Server 端，并不能判断哪个数据是由什么形式的页面元素提交的。换句话说，在页面中添加一个<input type="hidden" name="_action_Update" value=""/>，然后提交，Grails 一样也会根据_action_XXX 后面的 XXX 去确定由哪个 action 处理表单。Grails 的这一设计，也为程序员解决形如“根据用户选择的下拉列表框数据，选择不同的 action 进行处理”这样的需求提供了便捷。g:actionSubmit 可以用 action 属性指定接收请求的 action，此时可以用 value 属性指定按钮上显示的文本。

GSP 页面（views）中的表单已经构造完毕，那么 Controller 中的 action 又是如何接收表单数据的呢？进入 GoodsController 的 update action：

```
def update = {
    def goodsInstance = Goods.get( params.id )
    if(goodsInstance) {
        goodsInstance.properties = params
        if(!goodsInstance.hasErrors() && goodsInstance.save()) {
            flash.message = "Goods ${params.id} updated"
            redirect(action:show,id:goodsInstance.id)
        }
        else {
            render(view:'edit',model:[goodsInstance:goodsInstance])
        }
    }
    else {
        flash.message = "Goods not found with id ${params.id}"
        redirect(action:edit,id:params.id)
    }
}
```

这段代码实现了接收表单并更新数据库中的记录。首先讨论如何接收表单。params 是用于获取表单数据的 Map。使用 params.id 或者 params['id']，就可以取得 name 为 id 的表单项的值。

def goodsInstance = Goods.get(**params.id**)，这里用到了一个很重要的 Domain 方法：get 方法。get 可以实现用一个 id 值（主键值），去获取表中的一条记录。在取出了一条 goods 记录后，goodsInstance.properties = params 可以把 Map 中的键值按照同名的原则赋值给 goods 对象。如 params['category.id']的值，会赋给 goodsInstance 的 category 属性的 id 属性。对 properties 属性赋值，确实可以极大地帮助程序员减轻由表单构造对象的工作。

由于 Groovy 类都包含使用 Map 参数初始化对象的默认构造函数，因此还可以使用如下代码构造对象：def goodsInstance =new Goods(params)[1]。此时，goodsInstance 的内容已经被表单数据初始化了。

Domain 类的 hasError 方法可以检查记录是否符合约束条件，save 方法可以实现将记录保存到数据库。

---

[1] 从语义上讲，更新数据库中的数据，不应该使用 new 创建新的对象。从技术角度讲，使用 new 创建的对象不存在于 Hibernate 的持久化上下文中，因而无法实现更新。Grails 中推荐的更新数据库记录的作法是先用 get 方法从数据库读取数据，并构造 domain 对象的实例，然后再用 properties 属性去更新其内容。读者不必过分担心 get 请求带来的数据查询开销，Hibernate 的缓存机制可以解决这个问题。关于缓存的介绍，参见本书的第三部分。

Controller 的 redirect 方法和 render 方法都用于显示其他的页面。但它们有本质的区别：redirect 方法相当于在客户端执行跳转，会发起新的 HTTP 请求；而 render 方法，在于指定了下一步要处理显示逻辑的页面，在客户端并不存任何跳转。redirect 方法只能通过 url 传递简单类型的参数数据，而 render 方法可以通过 model 传递完整的数据供页面使用。

flash 相当于是 Controller 中提供的一个“通道”，它能够在当前 HTTP 请求和下一次 HTTP 请求间共享数据。在下一次 HTTP 请求完成时，会自动清除内容，Grails 帮助程序员控制了它的完整生命周期。

GSP 页面中可以直接访问 flash，显示 flash.message 内容的代码如下：

```
<g:if test="${flash.message}">
<div class="message">${flash.message}</div>
</g:if>
```

删除商品逻辑更加简单，GoodsController 中 delete action 的程序代码如下：

```
def delete = {
    def goodsInstance = Goods.get( params.id )
    if(goods) {
        goodsInstance.delete()
        flash.message = "Goods ${params.id} deleted"
        redirect(action:list)
    }
    else {
        flash.message = "Goods not found with id ${params.id}"
        redirect(action:list)
    }
}
```

一句话概括就是：根据 id 取出商品，当它存在时，调用 delete 方法删除记录，不存在时利用 flash 报告警告信息，最后，跳转到 list action 的页面。

4.4　本章小结

本章首先创建了应用程序并完成了数据库的配置。然后阅读并修改了 Grails 自动生成的代码，实现了商品列表和商品维护。介绍了数据输出、数据遍历、链接的制作，然后介绍了表单构造、表单接收、表单输出几个基本技术，同时还介绍了少量的数据库相关的操作（list 方法、get 方法、save 方法、delete 方法）。下面的章节将完成一些不能由 Grails 自动生成的功能，以进一步学习 Grails。表单相关的还有一个重要的知识点是表单验证，将在第 6 章进行详细的介绍。

从本章的内容中，读者还可以体会出使用 Grails 开发的一个特点，就是先用 Grails 自动生成脚手架代码，然后在它的基础上进行修改，这样做，不但开发速度快，而且程序质量高。

第5章 商品搜索

为了让顾客能够快速地找到他想要的商品，提供一个商品搜索的功能是十分必要的。这里希望用户能按照商品的名称、分类、商品描述以及价格区间进行搜索。本章要求读者掌握如下技能：自由地构造表单、进行复杂的数据库查询（***HibernateCriteriaBuilder***）、显示分页导航（***g: paginate***）。

5.1 构造查询表单

回顾一下前一章用于编辑商品的表单页面，是在 Grails 自动生成的 edit 页面上进行修改，因此很快就能完成相应工作。在本节中需要搭建一个用于搜索的页面，该页面可以在 edit 页面上进行改造。首先，在..\views\goods\中新建一个文件 searchForm.gsp，把 edit.gsp 的代码全部复制过来。然后，修改代码如下（这里仍然仅包含表单相关的内容，完整的代码读者可以参阅本书附带光盘的代码 ..\grails-app\views\goods\searchForm.gsp）：

```
<g:form method="get" action="search" >
Category: <g:select optionKey="categoryName" optionValue="categoryName"
from="${ Category.list() }" name="categoryName"
    noSelection="${['':'']}" ></g:select>
Title:  <input type="text" id="title" name="title" />
Description:
 <textarea id="description" name="description"></textarea>
Price:
    <input type="text" id="priceLow" name="priceLow" />
To:  <input type="text" id="priceHigh" name="priceHigh" />

<input type="submit" class="save" value="Search" />
</g:form>
```

g:select 的 optionKey 选择了 categoryName 而不是 id，是为了增加这个查询的实现难度，因为查询 categoryName 就需要对数据库的两张表进行联合查询。读者只有掌握了能解决更复杂问题的技术，处理实际问题才能更得心应手。

上面表单页面中的新知识点并不多，只要注意一下 g:select 的 noSelection 属性，它的作用是：当下拉列表框没有指定默认选项或指定了默认选项但该选项不存在时，选中下拉

列表框的默认选项。noSelection="${['':'']}"相当于加入了值和显示都为空字符串（""）的选项。

由于当前 GSP 页面仅用于输出一个查询表单，并不需要 Controller 提供数据，因此只需要在 GoodsController 中放一个空的 searchForm action 即可：

```
def searchForm = { }
```

运行程序并访问 http://localhost:8080/GDepot/goods/searchForm，会看到如图 5-1 所示的搜索页面。



图 5-1　搜索商品的表单

5.2　复杂的数据库查询

只用了很短的时间，就设计好了用于查询的表单页面，但这里的查询逻辑却并不简单：category 字段，如果选择了非空，则需要连接 Category 表，并用该字段进行比较；title 和 description 字段，如果不为空，需要用 like 语句进行判断；priceLow 和 priceHigh 分别指定价格的下限和上限，也是仅在其值不为空的时候有效……

需求的提出很自然，但解决起来却并不简单。如果用传统的字符串拼 SQL 语句的做法，可以想象一下，由于每个条件都不是必填的，仅仅判断该项是否需要对应的 where 子语句，拼接的逻辑就会非常复杂，更何况 category 还会决定是否要连接另外一张数据库表。

考虑到本节的代码可能需要反复调试，如果不希望在实际的页面上测试查询的逻辑，

可以使用 Grails 的控制台。相信读者不会忘记 Groovy 的控制台，那是一个非常实用的 Groovy 程序运行接口，在其中调试简单的代码是非常愉快的。在 Grails 中同样可以使用控制台。在命令行输入 grails console 命令即可启动 Grails 的控制台，如图 5-2 所示。

```
>grails console
```

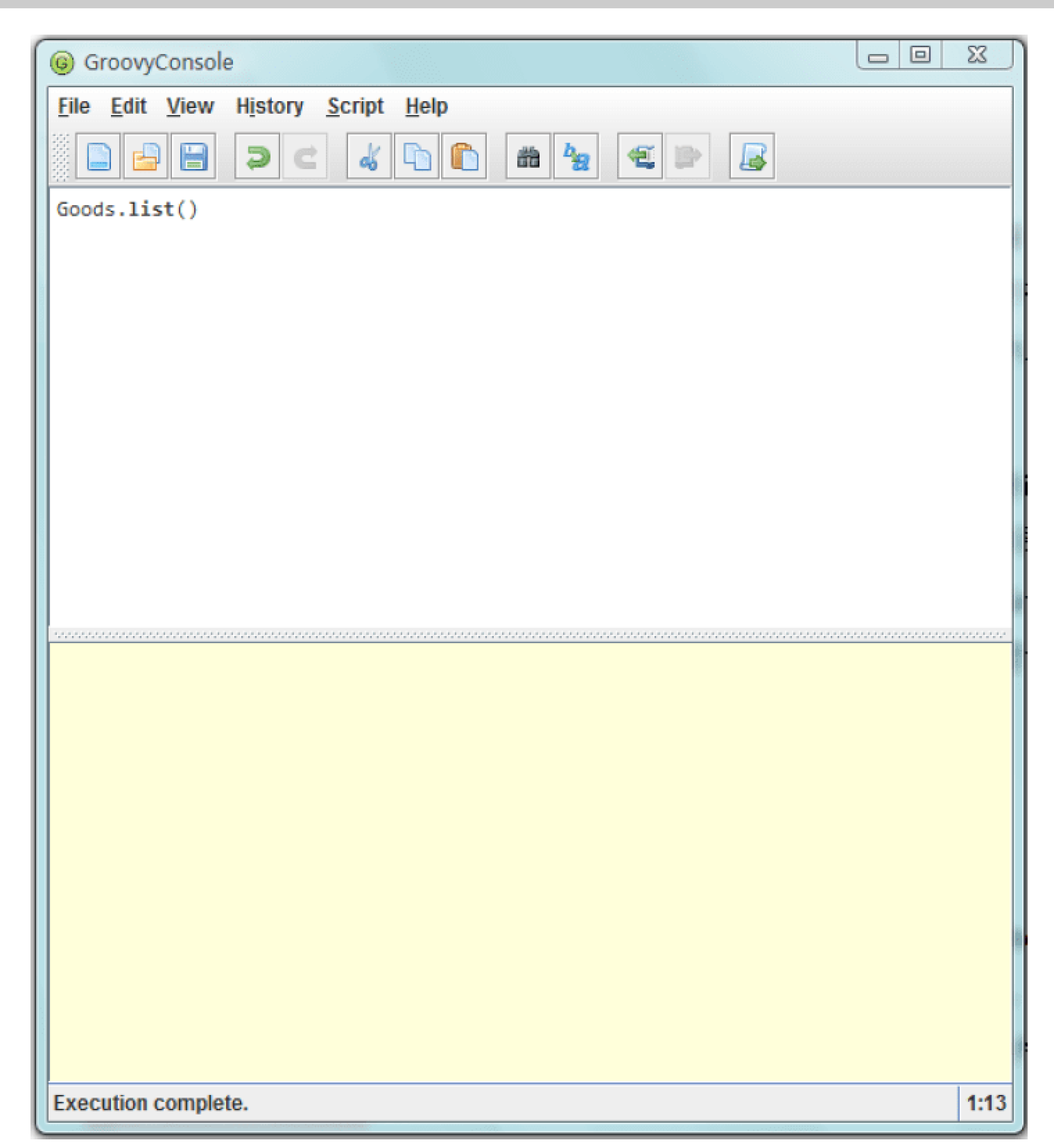


图 5-2　Grails 中的 GroovyConsole

本节编写的数据库查询的代码，推荐在控制台上运行并调试。不过，不要忘了存盘，不然修改 Domain 类等操作可能会导致控制台自动重启而丢失代码。

5.2.1　HibernateCriteriaBuilder 的初窥

针对这样复杂的查询问题，Hibernate 给程序员提供了很好的方案，那就是构建 Criteria 查询。它可以以一种很优雅的方式拼接出很复杂的查询逻辑。更加幸运的是，Grails 又对 Hibernate 的 Criteria 查询进行了更优雅的封装，Grails 为它提供的组件名为 HibernateCriteriaBuilder。

这里修改 GoodsController 的代码，加入一个 search action，如下所示：

```
def search = {
```

```
        def c = Goods.createCriteria()
        def goodsList = c.list {
            if(params.categoryName) {
                category{
                    eq('categoryName', params.categoryName)
                }
            }
            if(params.title) {
                like('title',"%${params.title}%")
            }
            if(params.priceLow) {
                ge('price',new BigDecimal (params.priceLow))
            }
            if(params.priceHigh) {
                le('price',new BigDecimal (params.priceHigh))
            }
            if(params.description) {
                like('description', "%${params.description}%")
            }
        }
        render(view:'list',model:[goodsInstanceList: goodsList])
    }
```

这部分代码对于读者来说，是全新的知识点，数据库的查询采用了 Grails 的 Hibernate CriteriaBuilder。首先用 Goods.createCriteria()创建了一个针对 Goods 的 Criteria 查询。然后将一个描述查询逻辑的闭包传给它的 list 方法，便实现了查询。而这个闭包的代码是十分优雅的，例如：

```
    if(params.categoryName) {
        category{
            eq('categoryName', params.categoryName)
        }
    }
```

这段代码表示，当表单提交的参数“categoryName”不为空字符串（''）时，与 category 表进行关联，并加入查询条件：category 表的 categoryName 要与 params.categroyName 相等。

类似地，if (params.title) { ***like('title',"%${params.title}%")*** }表示当 params.title 不为空时，加入查询条件：goods 表的 title 字段 like ***"%${params.title}%"***[1]。

同理，***ge('price',new BigDecimal (params.priceLow))*** 和 ***le('price',new BigDecimal (params.priceHigh))*** 表示加入 price 要大于等于 params.priceLow 和 price 要小于等于 params.priceHigh。

理解了上面的代码，相信读者一定会被 Grails 的 HibernateCriteriaBuilder 的优雅所感动。

---

[1] 这里要用 SQL 语句的 like，用于模糊查询，%表示通配符。

HibernateCriteriaBuilder 相当于是一种进行数据库查询的 DSL。显然，使用 DSL 可以更加优雅和轻松地完成原本复杂的任务。接下来对 HibernateCriteriaBuilder 的使用作更详细的介绍，表 5-1 包含了在 Criteria 中可用的方法（用于编写查询条件）。

表 5-1 Criteria 中可用的查询方法

名称	描述	样例
between	约束属性的值在给定的值之间	between("balance", 500, 1000)
eq	约束属性的值等于特定的值	eq("branch", "London")
eqProperty	约束某一属性必须等于另一属性	eqProperty("lastTransaction", "firstTransaction")
gt	约束属性的值大于特定的值	gt("balance",1000)
gtProperty	约束某一属性必须大于另一属性	gtProperty("balance","overdraft")
ge	约束属性的值大于或等于特定的值	ge("balance",1000)
geProperty	约束某一属性必须大于或等于另一属性	geProperty("balance","overdraft")
idEq	约束某一对象的 id 等于给定的值	idEq(1)
ilike	大小写敏感的 ilike 查询	ilike("holderFirstName","Steph%")
in	约束某一属性被包含于给定的 list 值中（注： in 是 groovy 的保留字，使用时需要加引号）	'in'("holderAge",[18..65])
isEmpty	约束属性的集合为空	isEmpty("transactions")
isNotEmpty	约束属性的集合不为空	isNotEmpty("transactions")
isNull	约束属性的值为 Null	isNull("holderGender")
isNotNull	约束属性的值不为 Null	isNotNull("holderGender")
lt	约束属性的值小于特定的值	lt("balance",1000)
ltProperty	约束某一属性必须小于另一属性	ltProperty("balance","overdraft")
le	约束属性的值小于或等于特定的值	le("balance",1000)
leProperty	约束某一属性必须小于或等于另一属性	leProperty("balance","overdraft")
like	等同于 SQL 里的 like 查询（大小写不敏感）	like("holderFirstName","Steph%")
ne	约束属性的值不等于特定的值	ne("branch", "London")
neProperty	约束某一属性不等于另一属性	neProperty("lastTransaction","firstTransaction")
order	按照某一特定属性来给查询结果排序	order("holderLastName", "desc")
sizeEq	约束属性集合的大小等于特定的值	sizeEq("transactions", 10)

默认的情况下，查询条件之间的关系是“与”的关系，可以通过 or、not 实现复杂的逻辑关系，如下例所示：

```
def searchClosure = {
    or {
        eq('field1', 'aaa')
        eq('field2', 'aaa')
    }
    eq('field3', 'aaa')
}
```

等价于如下的查询逻辑：

```
(fiedl1 = 'aaa' or field2 = 'aaa') and field3 = 'aaa'
```

5.2.2 数据库的分页查询

接下来回到之前的查询代码中，会发现还有一些问题没有解决。通常情况下，不能将全部查询结果直接返回到页面，因为有可能一次查询就产生非常海量的数据，若全部输出页面，一方面页面数据过多对用户也并不友好，另一方面将给服务器带来较大的负荷。因此，分页的数据库查询和分页的显示输出是必不可少的需求。

提到分页的功能，实际上有两件事情要做：一件事情是要进行分页的查询；另一件事情是在页面显示时能提供一个可以在不同页切换的分页导航。

要实现分页查询，HibernateCriteriaBuilder 提供了多种实现方法。在闭包中，可以使用 maxResults()和 firstResult()方法去控制分页的查询，具体操作过程如下：

```
if (!params.max) params.max = 10
if (!params.offset) params.offset = 0
def goodsList = c.list {
    if(params.categoryName) {
        category{
            eq('categoryName', params.categoryName)
        }
    }
    if(params.title) {
        like('title',"%${params.title}%")
    }
    if(params.priceLow) {
        ge('price',new BigDecimal(params.priceLow))
    }
    if(params.priceHigh) {
        le('price',new BigDecimal(params.priceHigh))
    }
    if(params.description) {
        like('description', "%${params.description}%")
    }
    maxResults(params.max as Integer)
    firstResult(params.offset as Integer)
}
```

但是事情还没有完，由于分页后还要在页面上显示总页数的信息，还需要查询数据库，以计算出符合条件的数据总数。如果使用传统的 SQL 语句来实现，相信读者一定会想到用 count()函数去实现。在 HibernateCriteriaBuilder 中，也可以使用 count，具体做法如下：

```
c = Goods.createCriteria()
```

```
def goodsCount = c.get {
    projections {
        rowCount()
    }
    if(params.categoryName) {
        category{
            eq('categoryName', params.categoryName)
        }
    }
    if(params.title) {
        like('title',"%${params.title}%")
    }
    if(params.priceLow) {
        ge('price',new BigDecimal(params.priceLow))
    }
    if(params.priceHigh) {
        le('price',new BigDecimal (params.priceHigh))
    }
    if(params.description) {
        like('description', "%${params.description}%")
    }
}
```

这里 c.get 表示只保留查询返回的第一条记录。projections 表示要进行组查询（group by 语句）。在 projections 中调用 properties 方法可以指定 SQL 中要 group by 的字段，使用 max、min、avg、count、sum 等函数则可以查询某一字段的最大、最小、平均、个数、求和等的值。

功能是已经完成了，但是坏味道（bad smell）已经出现了。两个查询的逻辑是何等相似，但为何要写两遍呢？重复代码出现了，这不是个好现象。

事实上，HibernateCriteriaBuilder 还提供了 count 方法，可以实现取查询结果总数[1]。而且这个 count 方法，会自动忽略查询闭包中指定的 maxResults 和 firstResult，这样就为闭包重用提供了可能。现在只要将代码改造为如下模样：

```
if(!params.max) params.max = 10
if(!params.offset) params.offset = 0
def searchClosure = {
    if(params.categoryName) {
        category{
            eq('categoryName', params.categoryName)
        }
    }
    if(params.title) {
```

[1] count 方法在官方的网站中并没有给出，笔者通过阅读 HibernateCriteriaBuilder 的源代码掌握了该点。感兴趣的读者可以在最后一部分，对 HibernateCriteriaBuilder 的源码进行讨论。

```
            like('title',"%${params.title}%")
        }
        if(params.priceLow) {
            ge('price',new BigDecimal (params.priceLow))
        }
        if(params.priceHigh) {
            le('price',new BigDecimal (params.priceHigh))
        }
        if(params.description) {
            like('description', "%${params.description}%")
        }
        maxResults(params.max as Integer)
        firstResult(params.offset as Integer )
    }

    def c = Goods.createCriteria()
    def goodsList = c.list( searchClosure )

    c = Goods.createCriteria()
    def goodsCount = c.count(searchClosure)
```

此时，将查询逻辑提取出来，取名为 searchClosure，然后分别用 HibernateCriteriaBuilder 的 list 方法和 count 方法对数据库进行查询。这样便有了一个已分页的 goods 列表和全部符合条件的 goods 总数。

现在的代码已经比较优雅了，但情况还可以向更好的方向继续发展。HibernateCriteria Builder 的 list 方法，本身就支持分页[1]。回顾一下前一章介绍的 list 方法：Goods.list (params)可以实现根据 params 里的 max 和 offset 参数去控制分页查询，此外，若 Map 中还包含 sort 和 order，则还可以实现控制查询时的排序（sort 用于指定排序字段，order 用于指定是升序排序 asc 还降序排序 desc）。HibernateCriteriaBuilder 的 list 方法，也支持分页 Map。可以进一步把 search 的代码改写为如下：

```
    def search = {
        if(!params.max) params.max = 10

        def searchClosure = {
            if(params.categoryName) {
                category{
                    eq('categoryName', params.categoryName)
                }
            }
            if(params.title) {
                like('title',"%${params.title}%")
            }
```

[1] 这一点官方网站中也没有给出，作者是通过阅读源代码掌握的。

```
            if(params.priceLow) {
                ge('price',new BigDecimal (params.priceLow))
            }
            if(params.priceHigh) {
                le('price',new BigDecimal (params.priceHigh))
            }
            if(params.description) {
                like('description', "%${params.description}%")
            }
        }

        def c = Goods.createCriteria()
        def goodsList = c.list( params, searchClosure )
        def goodsCount = goods.totalCount

        render(view:'list',model:[goodsInstanceList:goodsList])
    }
```

当调用 HibernateCriteriaBuilder 的 list 方法时，如果传入两个参数，且一个参数是 Map，则该 list 方法将进行分页查询。单页的大小（maxResults）和第一条记录位置（firstResult）分别由 Map 的 max 和 offset 两个 key 来指定。

此时返回的 goods 也不再是简单的 List 类型，而是由 Grails 提供的 grails.orm.PagedResultList 类的实例。这个类也实现了 java.util.List 接口，除了拥有 List 的全部特征之外，还多了一个 totalCount 属性。这个属性的值就符合查询条件的记录总数[1]。

如果希望了解查询逻辑被转换成了什么样的 SQL 语句（不过大可不必担心 Hibernate 生成 SQL 有错误，毕竟这是全世界开源爱好者监督下的最成熟的项目），可以通过配置查看 SQL 的 log[2]。

修改数据源的配置文件（grails-app\conf\DataSource.groovy），在 dataSource 节点中加入 logSql = true，如以下代码中的黑斜体所示：

```
dataSource {
    pooled = true
    driverClassName = "com.mysql.jdbc.Driver"
    username = "root"
    password = "mysql"
    logSql = true
}
```

然后重新启动 Grails，再次执行查询，就可以在控制台上看到 Hibernate 生成的 SQL 了（下面的 SQL 是在所有查询条件都不为空的情况下产生的）：

---

[1] 从源代码里可以看到，记录总数的查询过程与之前介绍的 count 方法比较相似。

[2] 如果是 Grails 的控制台调试查询代码，开启了 logSql 后，可以直接在控制台的输出中看到 SQL 语句，非常方便实用。

```
Hibernate:
    select
        count(*) as y0_
    from
        goods this_
    left outer join
        category category_a1_
            on this .category id=category a1 .id
    where
        (
            category_a1_.category_name=?
        )
        and this_.title like ?
        and this .price>=?
        and this_.price<=? limit ?
Hibernate:
    select
        this_.id as id0_1_,
        this .version as version0 1 ,
        this .category id as category3 0 1 ,
        this_.description as descript4_0_1_,
        this_.photo_url as photo5_0_1_,
        this_.price as price0_1_,
        this_.title as title0_1_,
        category a1 .id as id1 0 ,
        category a1 .version as version1 0 ,
        category_a1_.category_name as category3_1_0_
    from
        goods this_
    left outer join
        category category a1
            on this_.category_id=category_a1_.id
    where
        (
            category_a1_.category_name=?
        )
        and this .title like ?
        and this_.price>=?
        and this_.price<=? limit ?
```

从上面的 SQL 的 log 可以看到，Grails 一共执行了两个 SQL：用 select count(*)查询了记录总数；用“limit ?”对单页记录的数目进行限制[1]，从而实现了分页的查询。从生成的

[1] 不同的数据库，实现分页所需的 SQL 语句是不一样的。Hibernate 在生成 SQL 时，屏蔽了这一层的区别，使得程序员能够以一种统一的方式，对不同的数据库进行分页查询。

SQL 中可以看到，默认的情况下，是对 goods 表和 category 表进行了外连接操作（left outer join）。

Hibernate 生成的 SQL 默认使用外连接有它自身的道理，因为这里是对 goods 进行查询，如果不使用外连接，那么就会导致 category 为空的 goods 将永远不会被取出。

5.2.3 将查询改造为 inner join

虽然使用外连接有它的道理，但出于灵活考虑，不用外连接的情况也需要支持。可以通过调用 fetchMode 方法，设置对其他表抓取模式为主动抓取，从而实现主动连接其他表，例如：

```
import org.hibernate.FetchMode
...
if(params.categoryName) {
    fetchMode("category", FetchMode.EAGER)
}
```

其中的 FetchMode 类是 Hibernate 提供的一个类，需要导入才能使用（import org.hibernate.FetchMode）。

此时还需要编写查询条件，但实践证明，如下写法并不能正常工作：

```
eq("category.categoryName", params.categoryName)
```

Hibernate 自动生成的 SQL 语句，会给 category 起比较奇怪的别名（category_a1_），因此需要自己指定别名，才能给 category 表的字段设置查询条件。Criteria 类提供了一个设置别名的方法（createAlias），用于为关联查询的表设定别名。

事实上，HibernateCriteriaBuilder 的查询闭包中可以直接调用 Criteria 对象的成员方法，前面调用的 maxResults 和 firstResult 方法，实际上就是调用了 Criteria 类的 setMaxResults 和 setFirstResult 方法。因而，可以调用 createAlias 方法，为连接的 category 表设定一个别名，然后再用别名进行查询：

```
if(params.categoryName) {
    fetchMode("category", FetchMode.EAGER)
    createAlias('category','c')
    eq('c.categoryName', params.categoryName)
}
```

其中的 c.categoryName 就体现了用 category 的别名 c 访问其属性 categoryName。此时日志中记录的 SQL 语句就变成了如下样式：

```
Hibernate:
   select
      count(*) as y0_
   from
      goods this_
```

```
        inner join
            category c1_
                on this .category id=c1 .id
        where
            c1_.category_name=?
            and this_.title like ?
            and this_.price>=?
            and this .price<=? limit ?
    Hibernate:
        select
            this_.id as id0_1_,
            this_.version as version0_1_,
            this_.category_id as category3_0_1_,
            this .description as descript4 0 1 ,
            this_.photo_url as photo5_0_1_,
            this_.price as price0_1_,
            this_.title as title0_1_,
            c1_.id as id1_0_,
            c1 .version as version1 0 ,
            c1 .category name as category3 1 0
        from
            goods this_
        inner join
            category c1_
                on this .category id=c1 .id
        where
            c1_.category_name=?
            and this_.title like ?
            and this_.price>=?
            and this_.price<=? limit ?
```

从生成的 SQL 可以看出，left outer join 变成了 inner join，并且 category 表使用了别名 c1_（显然和指定的 c 有关系）。

数据库的查询到这里就算告一段落了，但显示查询结果的页面还有一些工作，接下来就去修改显示查询结果的页面。

5.3　显示分页导航

在 search action 中编写了查询数据库的程序，并且指定了使用 list.gsp 去显示搜索结果。就搜索结果而言，list.gsp 并不需要做改动，因为 search action 向 list.gsp 传递数据时使用了与 list action 一致的数据接口：Map 的 key 都是 goodsInstanceList。但是，用于**分页导航**的标签还需要修改。

Grails 提供了一个用于显示分页导航的标签："g: paginate"。这个标签会输出一系列的链接，有"上页"，"下页"，以及具体某一页，如"1"、"2"…使用 paginate 标签可以使用以下一系列属性。

（1）total（必填）——数据记录总数，用于计算分页页数（相当于上一节的 totalCount）。

（2）action（可选）——指定链接指向的 action，默认为当前 action（参考 g:link）。

（3）controller（可选）——指定链接指向的 controller，默认为当前 controller（参考 g:link）。

（4）id（可选）——指定链接的 id（参考 g:link）。

（5）params（可选）——包含请求参数的 Map（参考 g:link）。

（6）prev（可选）——显示"上页"按钮的文本，默认使用 messages.properties 的 default.paginate.prev 项（关于资源文件及 i18n 的知识，可以参考第 6 章）。

（7）next（可选）——显示"下页"按钮的文本，默认使用 messages.properties 的 default.paginate.next 项。

（8）max（可选）——单页记录的最大值，默认为 10。仅在 params.max 为空时有效。

（9）maxsteps（可选）——最多在导航中显示多少页，默认为 10。仅在 params.maxsteps 为空时有效。

（10）offset（可选）——指定链接的 offset（从第几条数据开始显示），仅在 params.offset 为空时有效。

修改 list.gsp 的 paginate 标签，将其从：

```
<g:paginate total="${Goods.count()}"/>
```

修改为：

```
<g:paginate total="${goodsInstanceList.totalCount}" params="${params}" />
```

关于 goodsInstanceList.totalCount，上一节已经作了解释。goodsInstanceList 是 grails.orm.PagedResultList 类的实例，通过它的 totalCount 属性，就能得到符合查询条件的数据记录的总数。

为了让 paginate 生成的导航链接能够继续按照当前的查询条件进行查询，需要用当前页面 params 设置 g:paginate 的 params 属性的值。

已经接近成功了，不过这样的代码还存在一个小问题：list.gsp 原本是 list action 的显示页面， list action 向页面传递的 goodsInstanceList 是由 Goods.list(params)方法返回的，并不是 grails.orm.PagedResultList 类的实例。所以还需要对 GoodsController 的 list action 进行简单的修改，让它返回 PagedResultList。代码修改情况为：

```
def list = {
if(!params.max) params.max = 10
[goodsInstanceList:
new grails.orm.PagedResultList(Goods.list(params),Goods.count())]
}
```

到这里总算大功告成了，可以享受一下劳动成果了。运行命令：

```
>grails run-app
```

然后访问 http://localhost:8080/GDepot/goods/searchForm（由于开发阶段只有两条数据，为了方便调试，将 search action 中的 params.max 设为了 1，此时一页只会显示一条数据，就可以看到分页和分页导航的效果了，如图 5-3 所示）。

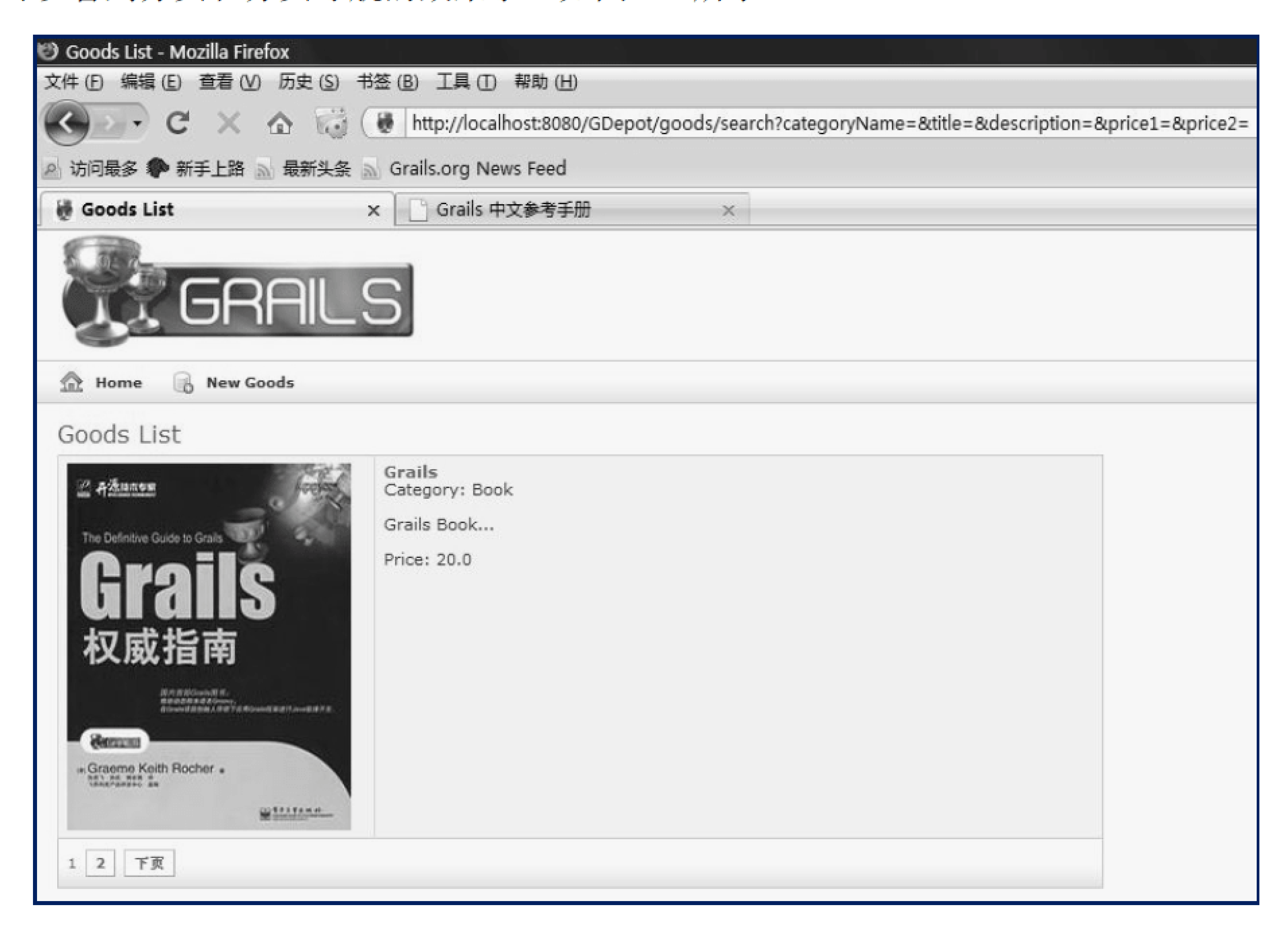


图 5-3 查询结果

5.4 本章小结

本章实现了商品搜索的功能，着重地介绍了如何使用 HibernateCriteriaBuilder 进行复杂的数据库查询，包括查询逻辑的组合拼接、组查询、分页查询等。相信读者掌握了 HibernateCriteriaBuilder 技术后，再去处理一般的数据库查询问题将会得心应手。

本章的最后一节介绍了通过 g:paginate 标签实现在页面上输出分页导航链接的方法。读者在学习了本章后，应该能够对一般性的“数据库查询+分页”问题拿出一个比较有效的实现方案了。

第6章 用户注册与登录

上一章实现了商品的搜索。本章将实现用户注册与登录。本章的知识点包括：表单验证、i18n 资源文件、Session 的使用、filter 类的使用。

6.1 表单验证与资源文件

要实现用户的注册与登录，首先要设计存储用户信息的 Domain 类。用 grails create-domain-class 命令创建一个 User Domain：

```
>grails create-domain-class User
```

然后修改 User.groovy 的代码如下：

```
class User {
    String userName
    String password
    String email
    String phone
    String address
    static constraints = {
        userName(size:2..10,blank:false)
        password(size:6..30,blank:false)
        email(email:true,unique:true,blank:false)
        phone(matches:/\d{7,11}/,blank:false)
        address(maxSize:200,blank:false)
    }
}
```

User 类中除了基本的字段属性，还包含了一个 static 的 constraints 闭包，其实在第 4 章商品维护中就见过这样的闭包。它在 Grails 的 Domain 中用于描述数据约束。通过 constraints，可以实现对数据的有效性进行检查，进而实现对表单数据的验证。Grails 中的 constraints 可以理解为一种进行数据约束的 DSL。

对于 User 类，这里限制它：

（1）userName 属性不能为空值，字符串长度为 2～20；

（2）password 属性不能为空值，字符串长度为 6～30；

（3）email 属性不能为空值，格式要符合 E-mail 的要求，而且在数据库中不能重复；

（4）phone 属性不能为空值，由 7～11 位数字组成；

（5）address 属性不能为空值，长度不超过 200。

理解了 User 类的约束条件，读者将发现 Grails 的约束描述非常简单和强大，只需要按照如下格式编写：

```
static constraints = {
    属性名(约束条件 1:约束值 , 约束条件 2:约束值 …)
}
```

Grails 中常用的约束条件如表 6-1 所示。

表 6-1　Grails 的数据约束条件

约束条件	描述	例子	错误代码
blank	用于验证 String 的值是否为空	login(blank:false) 设为 false 时，login 的值不能为空	className.property Name.blank
creditCard	用于验证 String 的值是不是有效的信用卡号码，内部调用 the org.apache.commons.validator.CreditCardValidator 类	cardNumber(creditCard:true) 设为 true 时，cardNumber 应该是一个有效的信用卡号码	className.property Name.creditCard.invalid
email	用于验证 String 的值是不是有效的 E-mail 地址，内部调用 org.apache.commons.validator.EmailValidator 类	homeEmail(email:true) 设为 true 时，homeEmail 应该是一个有效的 E-mail 地址	className.property Name.email.invalid
inList	用于验证 String 的值是否在一个范围或一个集合中	name(inList:["Joe", "Fred", "Bob"]) 约束 name 的值在给定的列表里	className.property Name.not.inList
matches	用于验证 String 的值是否符合给定的正则表达式	login(matches:"[a-zA-Z]+") login 的值需要符合正则表达式[a-zA-Z]+的条件	className.property Name.matches.invalid
max	确保一个值不超过给定的最大值	age(max:new Date()) price(max:999F)	className.property Name.max.exceeded
maxSize	确保一个值的范围不超过给定的最大值	children(maxSize:25)	className.property Name.maxSize.exceeded
min	确保一个值不小于给定的最小值	age(min:new Date()) price(min:OF)	className.property Name.min.notmet
minSize	确保一个值的范围不小于给定的最小值	children(minSize:25)	className.property Name.minSize.notmet
notEqual	确保属性的值不等于给定的值	login(notEqual:"Bob")	className.property Name.notEqual
nullable	当把 nullable 设置为 false 时，确保属性的值不能为 null	age(nullable:false)	className.property Name.nullable
range	确保属性的值出现在给定的范围中	age(range:18..65)	className.property Name.range.toosmall or className.property Name.range.toobig

续表

约束条件	描述	例子	错误代码
scale	给浮点数设置需要的刻度，如设置小数点右边的位数	salary(scale:2)	N/A
size	约束一个集合，一个数或一个字符串长度的大小范围	children(size:5..15)	className.propertyName.size.toosmall or className.propertyName.size.toobig
unique	约束一个属性在数据库里是唯一的	login(unique:true)，group(unique:'department')，login(unique:['group', 'department'])	className.propertyName.unique
url	用于验证一个字符串的值是否为有效的 URL	homePage(url:true)	className.propertyName.url.invalid
validator	用于给某一属性添加自定义的验证器		

表 6-1 的最后一列是错误代码，当验证发生错误时，可以根据错误代码输出错误信息。通过错误代码与资源文件的绑定，可以从中选择多国语言的错误描述。

接下来使用 grails generate-all 命令生成 User 的 CRUD 页面：

```
>grails generate-all User
```

用户注册本质上也是在创建用户，因此可以在 Grails 生成的 create 页面基础上进行修改。读者通过 create 页面可以体验一下 constraints 的实际效果，首先运行 run-app 命令：

```
>grails run-app
```

然后访问链接 http://localhost:8080/GDepot/user/create，随便填入一些用户信息后单击 create 按钮，可以看到如图 6-1 所示的效果。



图 6-1　默认页面的错误显示效果

阅读 Grails 生成的代码，可以帮助读者理解验证的执行过程。打开 UserController 的 save action，相信这一行代码一定会引起读者的注意（粗体+斜体的那行）：

```
def save = {
    def userInstance = new User(params)
    if(!userInstance.hasErrors() && userInstance.save()) {
        flash.message = "User ${userInstance.id} created"
        redirect(action:show,id:userInstance.id)
    }
    else {
        render(view:'create',model:[ userInstance:userInstance])
    }
}
```

每个 Domain 类中都包含一个用于记录错误信息的 errors 属性。userInstance.hasErrors()方法用于检查当前的 user 是否已经包含错误。此时的错误通常是表单提交时产生的类型转换错误，例如将页面上提交的字符串转换为日期类型，而这一转换可能因格式的不匹配而导致失败。当 userInstance.hasErrors()返回了假的时候，userInstance.save()会被执行。前面说过，save 方法是用于保存数据的，但它在保存之前，会先调用 validate 方法进行数据校验。若校验失败，则返回 null。

因此，用两句话总结 Controller 中进行表单校验的逻辑就是：先调用 hasErrors 方法，检查 errors 属性是否不包含错误，然后调用 validate 方法，进行校验；倘若该 Domain 实例需要存入数据库，则无需调用 validate 方法，直接调用 save 方法并判断返回值不为 null 即可。

Controller 中实现对 Domain 的验证，GSP 就要实现对错误信息进行输出。从图 6-1 中可以看到 Grails 自动生成的页面是以两种方式输出错误信息的：一种是将全部错误统一输出成一个错误列表；另一方式是判断或输出某一个属性的错误信息。

```
<g:hasErrors bean="${userInstance}">
<div class="errors">
<g:renderErrors bean="${userInstance}" as="list" />
</div>
</g:hasErrors>
```

上面的代码实现输出错误列表。其中 g:hasErrors 标签的功能与 Domain 的 hasErrors()方法类似，当 bean 属性值 user 有错误时，才执行标签内部的代码。g:renderErrors 标签用于输出错误信息的内容，as="list"表示以
的方式输出列表。如果想只输出某一属性的错误信息，则使用 field 属性，如 field="userName"，表示只输出 userName 属性相关的错误。当前 g:renderErrors 的 as 属性只支持 list 一种风格的显示效果，如果想要自己控制输出效果，则可以使用 g:eachError 标签。g:eachError 的使用与 g:each 类似，但仅用于遍历对象中的错误信息，取出错误信息后，可以由用户自己决定如何输出，例如：

```
<g:eachError bean="${ user }">
```

```
    <li>${it}</li>
</g:eachError>
```

或者：

```
<g:eachError bean="${ user }">
    <li><g:message error="${it}"/></li>
</g:eachError>
```

g:hasErrors、g:renderErrors、g:eachError 还可以一次处理多个对象。将多个需要处理的对象组装成一个 Map，然后将它传递给 model 属性。

```
<g:hasErrors model="${[user:user,cmd:cmd]}"...
```

此时的标签会同时检查 book 和 cmd 两个对象，看是否都没有错误。

Grails 生成的页面中将内容有误的输入项用红框圈了起来。这一点实现起来很简单，原理上是先判断这个属性是否有误，如果有，则对它所在的单元格使用显示有误内容的 CSS 样式表：

```
<td valign="top"
class="value ${hasErrors(bean:user,field:'userName','errors')}">
<input type="text" maxlength="10" id="userName" name="userName"
   value="${fieldValue(bean:user,field:'userName')}"/>
</td>
```

相信读者现在已经理解，验证数据和显示错误的程序代码了。但是，如何订制对不同错误显示不同的信息呢？

用资源文件可以完美地解决这个问题。回忆一下 2.2 节，grails-app\i18n 文件夹中存放了针对不同语言的资源文件。Grails 会自动根据当前的请求（客户端系统使用的语言），选择对应的资源文件，从而就实现了国际化的支持。而这与报错信息有没有关系呢？有关系，回忆一下表 6-1 的内容，每个约束条件都对应一个错误代码。在显示错误信息的时候，Grails 就是根据这个错误代码，将资源文件中对应项的内容显示出来。以 User 类的 userName 属性为例，在 message_zh_CN.properties 中加入如下内容：

```
user.userName.blank=用户名不能为空
```

然后再次尝试提交一个用户名为空值的表单，就会发现错误信息已经发生了相应的改变，如图 6-2 所示。

熟悉 Java 的读者可能会问，Java 的 properties 文件是不支持中文的，含有中文都需要先用 JDK 提供的 native2ascii 命令将其转换为 Unicode 编码才可以使用，为什么可以直接写中文呢[1]？事实上，是 Grails 在后台自动帮助完成了这个任务，从控制台的日志上可以看到这样的输出：

[1] 这里需要注意一点，message_zh_CN.properties 文件编码要指定为 UTF-8，不能使用 GBK。

Create User

[class User]类的属性[address]不能为空
[class User]类的属性[email]不能为空
[class User]类的属性[password]不能为空
[class User]类的属性[phone]不能为空
用户名不能为空

User Name:
Password:
Email:
Phone:
Address:

Create

图 6-2　自定义 userName 的错误信息

```
[native2ascii] Converting 1 file from D:\workspace\GDepot\grails-app\i18n
to C:\Users\Liang Shixing\.grails\1.0.4\projects\GDepot\resources\grails
-app\i18n
```

使用 message 方法，可以取出资源文件中的内容[1]。

例如：

```
${message( code:'userName' )}
```

或

```
<g:message code="userName" />
```

在资源文件 message_zh_CN.properties 中加入：

```
userName=用户名
```

将资源文件 message.properties 另存为 message_en.properties，并加入

```
username=User Name
```

当客户端系统默认语言为英文或在 URL 中加入参数 lang=en，页面上将显示“User Name”；相应地，中文系统或在 URL 中指定 lang=zh_CN 则会显示简体中文的“用户名”。理解了上面的例子，再用 Grails 创建提供多国语言支持的网站就不难了。

资源文件的内容还是可以传参数的。{0}就表示传入的第一个参数，{1}表示第二个，以此类推。

[1] GSP 中还可以使用 g:message 标签。

例如，在页面中编写：

```
${message ( code: "test_params", args:[1,2,3] )}
```

或

```
<g:message code="test_params" args="${[1,2,3]}"/>
```

并且在资源文件中编写：

```
test_params=数 1:{0},数 2:{1},数 3:{2}
```

运行时则输出：

```
数 1:1,数 2:2,数 3:3
```

一般来说，对 Domain 类验证时产生错误的信息都会传入参数：{0}为属性名，{1}为类名，{2}为属性值。

6.2 用户注册

上一节对 Grails 的表单验证和 i18n 资源文件做了介绍。细心的读者也许会问，如果表单验证完全是依赖于对 Domain 的验证，那么与 Domain 无关的表单怎么验证？就用户注册而言，需要提交的就不是 Domain，因为这里需要让用户输入两次密码，并校验这两次的输入是否一样。

在 Grails 中，除了可以用 Domain 进行表单验证，还可以用命令对象（Command Object）。它一样可以用 contraints 闭包去描述数据约束，一样可以用 hasErrors 方法去验证数据是否合法。唯一的不同是，它与数据库无关，因而不能执行数据库的相关操作。

命令对象的类名为 XXXCommand，它可以存放在 src\groovy 中，也可以直接写在 Controller 类中。在 src\groovy 中创建一个名为 RegisterCommand.groovy 的文本文件，然后加入如下内容：

```
class RegisterCommand{
    String password
    String passwordAgain
}
```

RegisterCommand 的目的在于让用户在注册时，输入两次密码，验证以保证两次输入的密码是一样的。因此，为 RegisterCommand 添加 constraints 闭包：

```
class RegisterCommand{
    String password
    String passwordAgain
```

```
        static constraints = {
            passwordAgain(validator: {
                    val, obj ->
                    if( obj.password != val)
                        return 'differentPassword'
            })
        }
    }
```

在 UserController 中创建一个空 action: def regester = {}，然后将 user\create.gsp 另存为 user\register.gsp。接下来修改 register.gsp 的内容，增加一个要求用户再次输入密码的文本框：

```
<tr class="prop">
   <td valign="top" class="name">
      <label for="password"><g:message code="password.again"/>:</label>
   </td>
<td valign="top"
class="value ${hasErrors(bean:cmd,field:'passwordAgain','errors')}">
      <input type="password" maxlength="30"
id="password" name="passwordAgain"
value="${fieldValue(bean:cmd,field:'passwordAgain')}"/>
   </td>
</tr>
```

由于现在可能包含错误信息的对象不局限于 User Domain，RegisterCommand 也可能包含，因此需要修改用于输出错误列表的代码为：

```
<g:hasErrors model="${[user:user,cmd:cmd]}">
<div class="errors">
   <g:renderErrors  model="${[user:user,cmd:cmd]}" as="list" />
</div>
</g:hasErrors>
```

最后将用于输入 password 的文本框的类型改为 password，将输入 address 的文本框改为多行文本框（textarea）。至此，页面的改造基本完成。

接下来修改 UserController 的 save action。用命令对象去接收表单更加容易，只需要将它写作闭包的参数：

```
def save = { RegisterCommand cmd -> }
```

此时，表单提交的数据就写入到了 cmd 中，并且数据也经过了验证。需要做的就是用 hasErrors()方法判断数据是否合法。

由于加入了 Command Object 用于检查两次输入的密码是否一致，原本的保存逻辑会发生一些改变，因此，修改代码如下：

```
def save = { RegisterCommand cmd -> //L1
    def user = new User(params)
    if(!user.hasErrors() && user.validate() && !cmd.hasErrors() ) { //L2
        if(user.save()) {
flash.message =
    message(code:'user.register.success',args:[user.userName] )
            redirect(controller:'goods',action:'list')
            return
        }
    }
    render(view:'register',model:[user:user,cmd:cmd])  //L3
}
```

有 3 行代码的修改值得关注。

L1：使用 Command Object 接收表单数据。

L2：首先将 user.save()改为 user.validate()，因为此时需要先保证!cmd.hasErrors()才能保存 user。但是，也不能写成 !user.hasErrors() ***&& !cmd.hasErrors()*** && ***user.save()***。因为当 cmd.hasErrors()为真时，user.save()将不再被执行，所以 user.validate()也没有执行，于是传递到页面的 user 也就不包含错误信息了。这里当然希望一次提交就尽可能多地发现错误，不会希望只发现一个“两次输入的密码不一致”的错误。

L3：由于 user 对象和 cmd 对象都可能包含错误信息，将这两个对象同时传给页面，供页面显示错误信息。

下一步，修改资源文件[1]：

```
user.userName.blank=用户名不能为空
user.userName.size.toosmall=用户名长度不能少于{3}个字符
user.userName.size.toobig=用户名长度不能超过{3}个字符
user.password.blank=密码不能为空
user.password.size.toosmall=密码长度不能少于{3}个字符
user.passwrod.size.toobig=密码长度不能超过{3}个字符
user.email.email.invalid=email 地址"{2}"格式不正确
user.email.unique=email"{2}"已注册，请使用别的 E-mail
user.email.blank=email 不能为空
user.phone.matches.invalid=电话"{2}"格式不正确
user.phone.blank=电话不能为空
user.address.maxSize.exceeded=地址长度不能超过{3}个字符
user.address.blank=地址不能为空
registerCommand.passwordAgain.differentPassword=两次输入的密码不一致
user.register.success=用户{0}注册成功
```

重新提交表单可以发现，当提交不正确数据时效果如图 6-3 所示。页面上还有一些元素没有进行汉化，不过相信读者现在已经能够轻松地解决国际化相关的任何问题了，这里

[1] 本书默认是修改简体中文的资源文件：message_zh_CN.properties。

就不再赘述。

Create User

地址不能为空
email"liangshixing@gmail.com"已注册，请使用别的email
密码不能为空
电话不能为空
用户名长度不能少于2个字符
两次输入的密码不一致

用户名: 1
password:
password.again: •
Email: liangshixing@gmail.com
Phone:
Address:

Create

图 6-3　用户注册的报错效果

当提交正确的数据后，如果用户保存成功，将自动跳转到显示商品列表的页面，并将给出注册成功的提示，如图 6-4 所示。

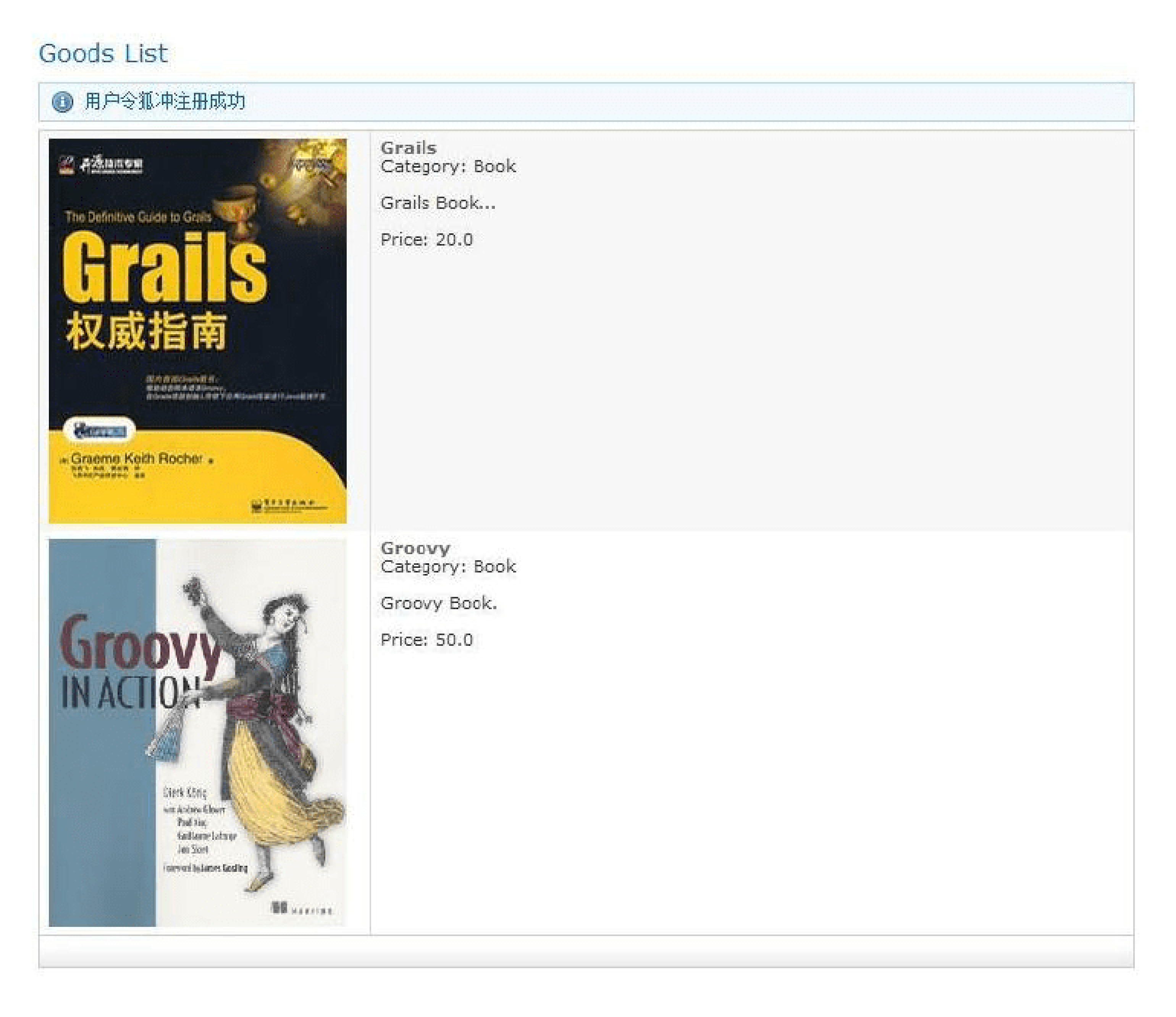


图 6-4　注册成功后跳转到商品列表页面

到目前为止，用户注册的功能已经基本实现，但还不够完善：因为用户的密码不应该直接以明文的形式保存在数据库中，而应该加密一下。下一节将完成此功能。

6.3 用户登录

用户登录页面的表单很简单，只有两个表单项：用户注册时使用的 E-mail 和密码：

```
<g:form action="loginCheck">
    email: <input type="text" name="email"/>
    password: <input type="password" name="password"/>
</g:form>
```

在 UserController 中，定义 loginCheck action，则这个 action 中需要实现如下逻辑：

（1）接收表单提交的 email 和 password；

（2）查询数据库中是否包含 email 和 password 都相符的 user 记录；

（3）如果有，则在 Session 中记录 user 的 id，否则，通过 flash.message 报错；

（4）登录成功后跳转。

6.3.1 登录的数据库查询

数据库有两个查询条件，读者可能会想用 HibernateCriteriaBuilder 去实现这个查询，但那样未免有点杀鸡用牛刀了。Grails 给用户提供了一个动态方法，findAllBy*。如果说 Domain 类的 list 方法、save 方法、get 方法能让人感到神奇，那么这个 findAllBy 方法一定会令人感到震惊。因为这个 findAllBy 方法的方法名是可变的。

如果想查询出 email 为 XX 的 User，可以用 User.findAllByEmail(XX)。如果想用 email 和 password 两个条件进行联合查询，就可以调用 User.findAllByEmailAndPassword(email, password)。findAllBy 方法还可以支持比较（大于、小于、大于等于、小于等于、不等于）查询，如：Goods.findAllByPriceLessThanEquals(priceHigh) 表示查找所有 price 小于等于 priceHigh 的记录。findAllBy 方法同样支持分页和排序：如果传入的最后一个参数为 Map，则 Map 中名为 max 和 offset 的 key 可以决定如何分页（与 HibernateCriteriaBuilder 的 list 方法中使用的 Map 完全一样）。

使用 Domain 类的 findAllBy 方法可以实现复杂的查询逻辑。典型的 findAllBy 方法可以写成样式：Domain 类名.findAllBy([属性名][比较方式][逻辑运算])?[属性名][比较方式]。其中比较方式包含：

（1）LessThan——小于给定值；

（2）LessThanEquals——小于等于给定值；

（3）GreaterThan——大于给定值；

（4）GreaterThanEquals——大于等于给定值；

（5）Like——与 SQL 的 like 等价；

（6）Ilike——与 like 类似，不过是英文字母大小写敏感的；

（7）NotEqual——不等于给定值；

（8）Between——在两个值之间，需要两个参数，大于小的且小于大的；

（9）IsNotNull——不为 null 值，无需参数；

（10）IsNull——为 null 值，无需参数。

逻辑运算包含：And 和 Or。

findAllBy 中只能包含两个查询条件，即只能包含一个逻辑运算符[1]。这是因为 findAllBy 方法本来就是用于解决简单的查询问题，当查询逻辑非常复杂的时候，程序员就会考虑使用 HibernateCriteriaBuilder 或者 HQL (Hibernate Query Language)。

与 findAllBy 类似的，还有 findBy 和 countBy 方法：findBy 方法只返回符合条件的第一条记录，countBy 方法会返回全部符合条件的记录的总数。在解决登录问题的时候，用 findBy 刚好合适，因为这里只需要一条查询结果。

6.3.2 使用 Session 维持会话

接下来介绍 Grails 中 Session 的使用。

Session 在计算机中，尤其是在网络应用中，称为“会话”。

对于计算机科学领域，在特殊的网络领域，Session 是一种持久网络协议，在用户（或用户代理）端和服务器端之间建立关联，从而起到交换数据包的作用机制，Session 在网络协议（例如 Telnet 或 FTP）中是非常重要的部分[3]。

Grails 中 Session 的使用与 Map 相似：可以用“.”和“[ ]”两种方式访问其数据。如：session.userId=user.id 和 session["userId"]=user.id 都表示把 user.id 保存到 session.userId 中；调用 session.userId 或者 session["userId"]则可以将 Session 中的 userId 内容取出；如果需要清空 Session 的内容，可以通过 session.invalidate()方法实现。

基本知识介绍到这里，编写 loginCheck action 代码如下：

```
def loginCheck = {
    def user = User.findByEmailAndPassword(params.email, params.password )
    if(user) {
        session.userId = user.id
        flash.message =
            message(code:'login.success',args:[user.userName])
        redirect(action:'list',controller:'goods')
    } else {
        flash.message = message(code:'login.failed')
        redirect(action:'login')
    }
}
```

相应地，退出系统的代码如下：

---

[1] 这点限制可能会在新的版本中得到改进，可以参阅：http://jira.codehaus.org/browse/GRAILS-1186。

```
def logout = {
    session.invalidate()
    redirect(action:'login')
}
```


6.3.3 自定义 Codec 实现对密码加密

到目前为止，用户注册和用户登录功能已经基本完成，但在上一节结尾时曾提到，需要将用户的密码加密之后再保存到数据库中。在 Groovy 中要实现加密并不困难，可以直接使用 Java 的类库实现：

```
MessageDigest md = MessageDigest.getInstance('SHA')
md.update(user.password.getBytes('UTF-8'))
user.passord = (new BASE64Encoder()).encode(md.digest())
```

虽然对密码加密只需要 3 行代码，但在注册和登录都需要调用，因而最好将其封装成一个工具方法。回忆一下表 2-1 中 Grails 生成的项目目录结构。Grails 中约定编写工具方法类的文件夹是 utils。因而，在 utils 文件夹中创建一个名为 PasswordCodec.groovy 的文件，其内容如下：

```
import java.security.MessageDigest
import sun.misc.BASE64Encoder

class PasswordCodec {
   static encode = {str ->
      MessageDigest md = MessageDigest.getInstance('SHA')
      md.update(str.getBytes('UTF-8'))
      return (new BASE64Encoder()).encode(md.digest())
   }
}
```

这里是按 Grails 中的约定实现自定义编码类（XXXXCodec 类）。类名 PasswordCodec 是 Grails 的一种约定：一定要写成 XXXCodec，并且约定它一定要包含一个名为 encode 的闭包。约定了这么多内容，目的很简单，为了调用更加容易。定义了上面的类后，可以通过如下调用实现加密：

```
user.password = user.password.encodeAsPassword()
```

确实很神奇。定义了 utils 类，就可以在任意的对象上调用 encodeAsXXX 方法。这在传统的静态语言编程中是不可想象的，完全是得益于 Groovy 的 MOP 编程技术，更多关于 Groovy 的 MOP 内容，会在本书的最后一部分进行讨论。

有了这个 PasswordCodec 类，用户注册和登录都需要先对密码进行 encode。因此，将注册和登录的代码修改为：

```
def save = { RegisterCommand cmd ->
def user = new User(params)
    if(!user.hasErrors() && user.validate() && !cmd.hasErrors() ) {
        user.password = user.password.encodeAsPassword()
        if(user.save()) {
            flash.message =message(code:'user.register.success',args:
            [user.userName] )
            redirect(controller:'goods',action:'list')
            return
        }
    }
    render(view:'register',model:[user:user,cmd:cmd])
}
def loginCheck = {
    def user = User.findByEmailAndPassword(params.email,
                        params.password.encodeAsPassword() )
    if(user) {
        session.userId = user.id
        flash.message = message(code:'login.success',args:[user.
        userName])
        redirect(action:'list',controller:'goods')
    } else {
        flash.message = message(code:'login.failed')
        redirect(action:'login')
    }
}
```

6.4 登录保护

对于需要用户“先登录，后访问”的资源，可以在相应的 action 中加入这样的代码：

```
if(session.userId) {
    //已经登录
} else {
    redirect(action: 'login',controller: 'user')
}
```

但是通常情况下，有这样需求的页面并不在少数，甚至在比例上都不在少数。如果每个 action 都重复写着这样的代码，会带来大量的重复代码，严重影响程序的“美感”。针对这个问题，Grails 提供了两种解决方案。

（1）方案一：使用 Grails 的 beforeInterceptor 拦截器。

Grails 提供了两种拦截器，分别是 beforeInterceptor 和 afterInterceptor。它们的作用是在 action 执行之前或之后自动调用。因此，可以在 beforeInterceptor 中进行登录检查，并且

在需要的情况下，执行跳转。beforeInterceptor 有两种写法，一种是直接定义为一个闭包。此时，controller 中的任何 action 都会被拦截：

```
def beforeInterceptor = {
    //在其他 action 执行前，做一些事情
}
```

另一种方式，将 beforeInteceptor 定义为一个 Map。这种方式比较灵活，配置性也较强：

```
def beforeInterceptor = [action: 拦截闭包,
except:[不希望被拦截的 action 列表（其他会被拦截）],
only:[希望被拦截的 action 列表（其他不会被拦截）] ]
// except 和 only 不能同时使用
```

以当前的程序为例，可以通过如下代码实现登录保护：

```
def auth = {
    if(!session.userId) {
        redirect(action: 'login',controller: 'user')
        return false
}
}

def beforeInterceptor = [ action: auth,
except:['register', 'save', 'login', 'loginCheck', 'logout'] ]
```

若用户未登录时调用拦截器，则跳转到登录页面，并且拦截闭包会返回 false。如果拦截闭包返回 false，则被拦截的 action 将不再被执行，这就达到了保护登录的目的。

然而，上面的程序还有一个小问题：由于在 Controller 中定义了 auth 闭包，则 auth 也会被当作一个 action 来处理，可以通过 url: GDepot/user/auth 访问到它。这并不是这里想要的。为了解决这个问题，把 auth 定义成一个方法，在定义 beforeInterceptor 的时候用 this.& 将 auth 方法转换为闭包：

```
def auth() {...} //现在 auth 是方法，不是闭包
def beforeInterceptor = [ action: this.&auth,
except:['register', 'save', 'login', 'loginCheck', 'logout'] ]
```

（2）方案二：使用 Grails 的 filter 技术。

拦截器使用虽然简单，但也存在一定的不足，它只能作用于单一的控制器。若要在更大范围内进行拦截操作，可以选择 filter。

典型的 filter 类的定义模式如下：

```
class SecurityFilters {
    def filters = {
        filter1(controller: '*', action: '*') {//①
            before = {
```

```
                //先于 action 执行
            }
            after = {
                //后于 action 执行
            }
            afterView = {
                //在 view 输出后才执行
            }
        }
        filter2(uri: '/**') {//①
            before = {
                //先于 action 执行
            }
            after = {
                //后于 action 执行
            }
            afterView = {
                //在 view 输出后才执行
            }
        }
    }
}
```

Filter 类要存放在 grails-app\conf 文件夹下面，它的类名一定是 XXXFilters，类中一定要包含一个名为 filters 的闭包。在①这一级别，如 filter1，可以指定 filter 的作用范围：通过 controller 和 action 或者通过 uri，可以确定当前 filter 将拦截哪些 url。在 filter1 内部，可以选择定义 3 种拦截方式，分别是 before、after 和 afterView。与拦截器类似，如果在 before 闭包内返回 false，被拦截的 action 将不再执行。对于当前的登录保护，可以使用如下代码实现保护：

```
class SecurityFilters {
   def filters = {
      loginCheck(controller:'user', action:'*') {
         before = {
           if(!session.userId &&
                !actionName in ['login', 'loginCheck',
           'logout','register', 'save']) {
                redirect(controller: 'user',action:'login')
                return false
           }
         }
      }
   }
}
```

使用 beforeInterceptor 或者 filter 可以实现简单的登录保护，若需要进行更为复杂和精细的权限控制，可以使用 JSecurity 或 Acegi 等 Grails 插件来实现，这部分内容将在后面的章节里进行讨论。

6.5 本章小结

本章结合用户注册、用户登录、退出，以及登录保护这些功能的实现过程，介绍了 Grails 中表单验证、i18n 资源文件支持、Command Object、数据库查询的 findAllBy*系列方法、Session 的使用、beforeInterceptor 拦截器以及 filters 的使用。下一章将结合前几章所学的知识点，完成购物车及订单的提交与查看。

第 7 章 购物车与订单

本章将开发购物车与订单相关的功能，对前面各章的知识点进行综合应用。此外，通过本章读者还会学习到：**Service 类的使用、模板页面的使用**。

7.1 购物车的查看与管理

7.1.1 定义购物车的 Domain 类

现在开发购物车的功能，允许用户把商品添加到购物车中。首先需要设计 Domain 类。Cart 类是购物车类，LineItem 类是购物车中的商品记录

```
class LineItem {
    Goods goods                             //进入购物车的商品
    int itemNumber = 1                      //购买的数量
    static belongsTo = [cart:Cart]  //当前 LineItem 属于哪个购物车
    static constraints = {
        itemNumber(min:1)
    }
}
```

一条 LineItem 记录包含一件商品，并记录它的购买数量。static belongsTo = [cart:Cart] 指定了 LineItem 的属主（owner）是 Cart，并可以通过 cart 属性去访问它的 owner。指定了 owner 就意味着对 LineItem 实现了级联删除：当某条 cart 记录被删除时，所有属于它的 lineItem 都将自动删除。如果存在多个 owner，可以在 Map 中用逗号分隔，如：

```
static belongsTo = [owner1:Owner1Classname, owner2: Owner2Classname, …]
```

接下来设计 Cart 类如下：

```
enum CartStatus{NEW,ORDERED}
class Cart {
    static hasMany = [lineItems:LineItem]
    static belongsTo = [user:User]
    CartStatus status = CartStatus.NEW
```

```
    public def totalPrice() {
        def price = 0
        lineItems.each{ price += it.goods?.price * it.itemNumber }
        return price
    }
}
```


定义 Cart 类前，先定义了 CartStatus 枚举，用于标识购物车的两个不同的状态：“新购物车”和“已下单”。由 CartStatus 定义的 status 属性用于定义购物车的状态，并指定默认值为 CartStatus.NEW。查看 Grails 生成的数据库表结构，会发现 Domain 的枚举类型在数据库中被映射为数字类型。

```
CREATE TABLE 'cart' (
  'id' bigint(20) NOT NULL auto_increment,
  'version' bigint(20) NOT NULL,
  'status' int(11) NOT NULL,
  'user id' bigint(20) NOT NULL,
  PRIMARY KEY  ('id'),
  KEY 'FK2E7B20F7634DFA' ('user_id'),
  CONSTRAINT 'FK2E7B20F7634DFA' FOREIGN KEY ('user_id') REFERENCES 'user'
  ('id')
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
```

static hasMany 用于定义“一对多”关系。static hasMany=[lineItems:LineItem]指定了 Cart 与 LineItem 是“一对多”的关系，并且可以通过 lineItems 属性访问 Cart 的多个 LineItem。Cart 类还定义了一个 totalPrice 方法，用于计算购物车中商品的价格总和。

设计 Cart 的 status 属性，需要默认两个事实：

（1）只能向 status 为 NEW 的 Cart 添加商品；

（2）每位用户最多只能有一个状态为 NEW 的 Cart。

这两点需要用程序去保证，因此，读取出当前用户购物车的代码如下（该代码的创建将在下一小节介绍）：

```
def cart = null
if(session.userId) {
    def user = User.get(session.userId)//取出当前用户
    if(user) {
        //以 user 和 status 为条件去查询 cart
        cart = Cart.findByUserAndStatus(user,CartStatus.NEW)
        if(! cart) {
            //当不存在符合条件的 cart 时，创建并保存
            cart = new Cart(user:user, status: CartStatus.NEW)
            cart.save()
        }
    }
}
```

上面的代码可以实现为当前用户找出 Cart，因为逻辑比较复杂，所以效率不够理想，需要进行适当的优化。设想，如果在 session 中保存了当前用户购物车的 id，再次读取 Cart 将变得十分简单和高效。将上面的代码进行优化如下：

```
def cart = null
if(session.cartId) {
    cart = Cart.get(session.cartId)
    if(cart.status != CartStatus.NEW) cart == null
}
if (!cart && session.userId) {
    def user = User.get(session.userId)      //取出当前用户
    if(user) {
        //以 user 和 status 为条件去查询 cart
        cart = Cart.findByUserAndStatus(user,CartStatus.NEW)
        if(! cart) {
            //当不存在符合条件的 cart 时，创建并保存
            cart = new Cart(user:user, status: CartStatus.NEW)
            if(cart.save())
                session.cartId = cart.id    //将 cart.id 保存到 session 中
        } else {
            session.cartId = cart.id        //将 cart.id 保存到 session 中
        }
    }
}
```

利用 session 去缓存 cart.id 可以提高查找 cart 的效率，但不要忘了一点，那就是，当购物车状态发生变化时，应该将 session 中的 cartId 清除掉。这是之前默认的两个事实所决定的。

7.1.2 定义 OrderService 类

由于所有需要与购物车打交道的业务，都需要先调用这段代码，这样就十分有必要将它封装起来。Grails 可以通过创建服务（Service），帮助程序员把可重用的业务逻辑封装起来。执行 Grails 的 create-service 命令，可以创建 Service：

```
>grails create-service order
```

可以看到 Grails 在 grails-app\services 文件夹中，创建了一个名为 OrderService 的类。了解 Spring 的读者，应该对这种命名方式不会陌生。Service 类会被 Spring 的容器所托管，从而拥有了“依赖注入”与“事务处理”等特性[1]。

如果在 Service 类中定义 static transactional = true，则表示这个 Service 类的每个方法，

[1] 依赖注入 DI 与控制反转 IoC 是同一个概念，读者可以访问 www.springframework.org 去了解相关的内容，不了解这些概念不会影响学习这一章的知识。

都要被当成一个事务去处理[1]。这正是这里想要的，因此，修改 OrderService 的内容如下：

```
class OrderService {
    static transactional = true
    def prepareCart(session) {
    def cart = null
        if(session.cartId) {
             cart = Cart.get(session.cartId)
             if(cart && cart.status == CartStatus.NEW )
                return cart
        }
        if(session.userId) {
            def user = User.get(session.userId)
            cart = Cart.findByUserAndStatus(user,CartStatus.NEW)
            if( cart) {
                session.cartId = cart.id
                return cart
            } else {
                cart = new Cart(user:user,status:CartStatus.NEW)
                if(cart.save()) {
                    session.cartId = cart.id
                    return cart
                }
            }
        }
        return cart
    }
}
```

在 OrderService 类中定义了 prepareCart（session）方法，调用这个方法，可以得到当前用户的购物车。以传统的开发 Spring 应用程序的经验，要避免将 session 这样的东西传入到处理业务逻辑的 Service 类中。因为那样会使 Service 类与 Web 容器耦合在一起，从而给自动化测试或项目移植带来了困难。但是，在 Grais 中不用担心这个问题。在定义 prepareCart 方法时仅指定了参数的名称，并没有指定参数的类型。因此在调用 prepareCart 时，只需要传入一个与 session 用法相似（例如 Map，都可以用“.”和“[]”访问数据成员）的对象即可。这也是动态语言编程的一大特色。

由于 Service 类的对象是由 Spring 容器所托管的，因此调用 Service 类非常简单。可以使用 Spring 的依赖注入机制将其注入到需要调用它的类中。例如，现在需要在 GoodsController 中调用 OrderService 类，则只需要在 GoodsController 中添加一个名为 orderService 的属性：

```
class GoodsController {
```

[1] 事务是应用程序中一系列严密的操作，所有操作必须成功完成，否则在每个操作中所做的所有更改都会被撤销。事务具有原子性，一个事务中的一系列的操作要么全部成功，要么一个都不做。

```
    def orderService        //需要将 OrderService 的类名首字母小写
    …
}
```

Spring 容器会自动将容器中的 orderService 实例传入到 GoodsController 中，而无需对它进行初始化，这就是所谓的依赖注入。并且，Spring 不单能够向 Controller 中注入 Service 实例，在 Command Object、Taglib 类、单元测试类，或者其他 Service 类中，只要定义了与 Service 类名相同的属性，Spring 都会自动注入可用的实例。这里要强调一点，不要尝试自己用 new 初始化 Service 类。因为这样创建的 Service 实例没有运行在 Spring 容器中，它的事务管理特性和依赖注入特性是不可用的。

7.1.3　显示购物车

接下来要在 goods\list.gsp 页面显示购物车。要实现这个功能，需要先在 Controller 中取出购物车的数据。问题又来了，GoodsController 中有两个 action 最终会调用 list.gsp 进行显示，要在两个 action 中都加入取购物车的代码吗？回忆一下登录保护那一节的知识点，可以在 afterInterceptor 中做这件事情，因为在 afterInterceptor 中可以修改 action 传给 GSP 的数据。具体的代码如下：

```
def afterInterceptor =
    [action:this.&addCartToModel, only:['list','search'] ]
private def addCartToModel(model){
    model.cart = orderService.prepareCart(session)
}
```

在 list.gsp 中加入如下代码用于显示购物车：

```
<div>
    <g:if test="${cart?.lineItems}">
    <h1><g:message code="yourCart"/></h1>
    <table>
        <thead>
            <th><g:message code="goods.title" /></th>
            <th><g:message code="goods.price" /></th>
            <th><g:message code="lineItem.itemNumber" /></th>
            <th></th>
        </thead>
        <tbody>
        <g:each in="${cart?.lineItems}" var="lineItem">
            <tr>
                <td>${lineItem.goods?.title}</td>
                <td>
                ${lineItem.goods?.price} * ${lineItem.itemNumber}
                </td>
            </tr>
```

```
            </g:each>
            </tbody>
            <tr>
                <td colspan="2">
                <g:message code="cart.totalPrice" />
                ${cart.totalPrice()}
                </td>
            </tr>
        </table>
        </g:if>
        <g:else>
            <h2><g:message code="cart.empty"/></h2>
        </g:else>
    <div>
```

上面的代码对 cart.lineItems 进行遍历，然后输出每一条 lineItem 所引用的 goods 的信息，以及 lineItem 中记录的商品条数。对于传统的数据库编程，实现这种通过 Cart 找到 LineItem，再通过 LineItem 去访问 Goods，都需要使用表连接。但此时只是利用了 Domain 之间的关联，就实现了多表的数据访问。这是因为 Hibernate 有一个延时加载（Lazy load）的特性，利用这个特性，程序的开发会变得简单许多。

然而，延时加载有可能带来性能的问题，因为使用延时加载有可能会导致执行 N+1 次查询。以上面的代码为例，取 cart.lineItems 的时候，需要进行一次数据库查询。假设取出了 N 条记录，在遍历这 N 条记录的时候，再分别去取每条 LineItem 的 Goods，则此时会对 Goods 表进行 N 次查询。为解决这样的性能问题，Hibernate 提供了两种方案：一种是设置 fetchMode，使之在一次查询中完成抓取；另一种是使用缓存，使得 N 次查询都在缓存中进行，从而使得数据库的查询变为 1 次查询。关于 fetchMode 和缓存相关的内容，将在第 11 章进行讨论。

7.1.4 维护购物车

接下来开发“向购物车添加商品”的代码。在商品列表的表格中添加一列：

```
<td>
<g:form method="post">
<input type="hidden" name="goodsId" value="${goodsInstance.id}">
<g:actionSubmit action="addToCart"
value="${message( code:'cart.addToCart')}" />
</g:form>
</td>
```

在这一列中添加了一个按钮，单击按钮会向 addToCart action 提交当前商品的 id。接下来在 Controller 中实现 addToCart。

addToCart 的业务逻辑如下：

（1）首先，应取出当前用户的购物车；

（2）判断当前购物车中是否包含当前商品；

（3）如果包含，则对 lineItem 的 itemNumber 属性值加 1；

（4）如果不包含，则添加一条 lineItem，且指定它的 itemNumber 为 1。

因为 addToCart 也相对比较复杂，所以将这部分代码封装到 OrderService 中，添加如下方法：

```
def addToCart(session,goodsId) {
    def cart = prepareCart(session)//取出购物车
    if(cart) {
        def goods = Goods.get(goodsId)
        if(goods){
            def lineItem = null
            //判断购物车中是否包含此商品
            lineItem = LineItem.findByCartAndGoods(cart,goods)
            if(lineItem) {
                //如果包含，数量加 1
                lineItem.itemNumber = lineItem.itemNumber + 1
                lineItem.save()
            } else {
                //如果不包含，添加一条新记录，新记录的数量为 1
                lineItem =
                    new LineItem(cart:cart, goods: goods , itemNumber: 1)
                lineItem.save()
            }
            return lineItem
        }
    }
}
```

此时 GoodsController 中的代码就简单多了，可写为：

```
def addToCart = {
   if(session.userId && params.goodsId) {
    orderService.addToCart(session,params.goodsId)
   }
   redirect(action:list)
}
```

运行一下程序，单击“添加到购物车”按钮，就可以把商品添加到购物车当中，如图 7-1 所示。

接下来开发“从购物车移除”的功能。

首先修改页面，在显示购物车的表格中添加一列：



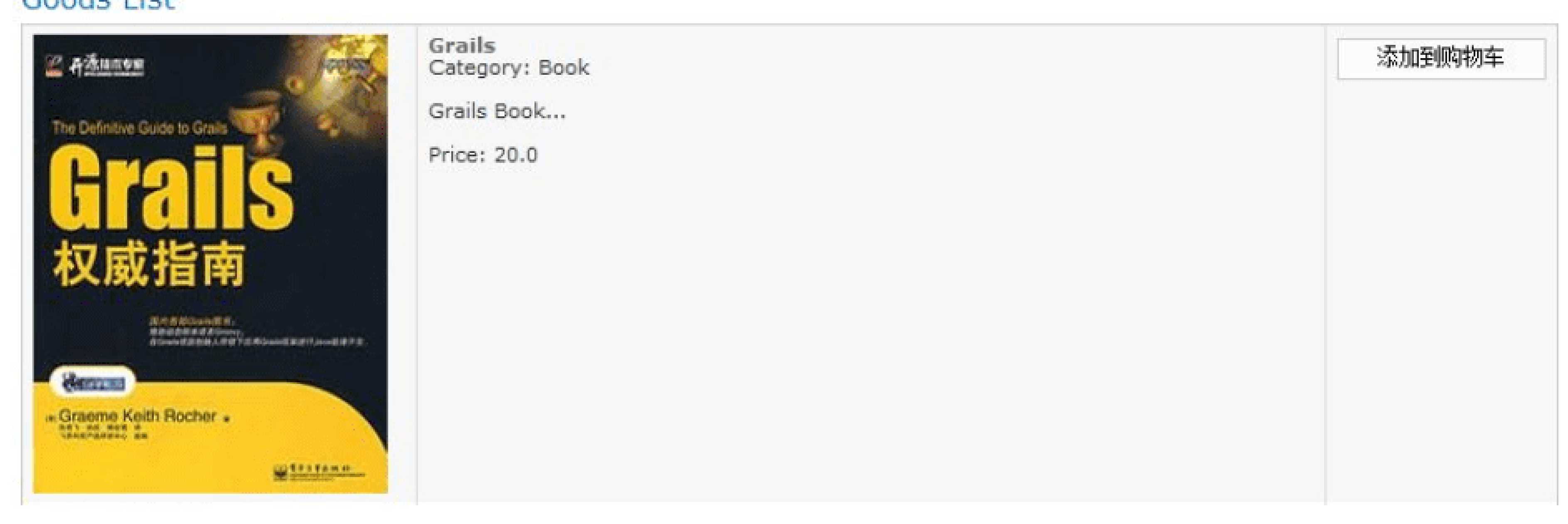


图 7-1 实现将商品添加到购物车

```
<td>
<g:form action="removeFromCart" method="post">
    <input type="hidden" name="id" value="${lineItem.id}">
    <input type="submit" value="${message (code:'cart.remove')}"
    onClick="return onfirm('${message(code:'confirm.
    removeFromCart')}')">
</g:form>
</td>
```

添加按钮，单击后会向 GoodsController 的 removeFromCart action 提交表单，表单中包含了要被删除的 lineItem 的 id。removeFromCart action 中需要的事情很明确，即根据表单提交的 id 值，删除相应的 lineItem：

```
def removeFromCart = {
    def lineItem = LineItem.get(params.id)
    if(lineItem) {
        lineItem.delete()
    }
    redirect(action:list)
}
```

上面的代码能够实现删除购物车中的商品，但它有一个不小的安全隐患。它没有判断要删除的 lineItem 是否真的属于当前用户的购物车，恶意的用户可以利用这个漏洞去删除任意的一条 lineItem，这将给整个系统带来严重的危害。因此，需要改进一下程序：

```
def removeFromCart = {
    def lineItem = LineItem.get(params.id)
    // 防止用户随意提交 id
    if(lineItem && lineItem.cart.id == session.cartId) {
    lineItem.delete()
```

```
        }
        redirect(action:list)
    }
```

刷新一下页面，单击“从购物车移除”按钮，弹出一个提示框，询问是否移除商品，如图 7-2 所示。

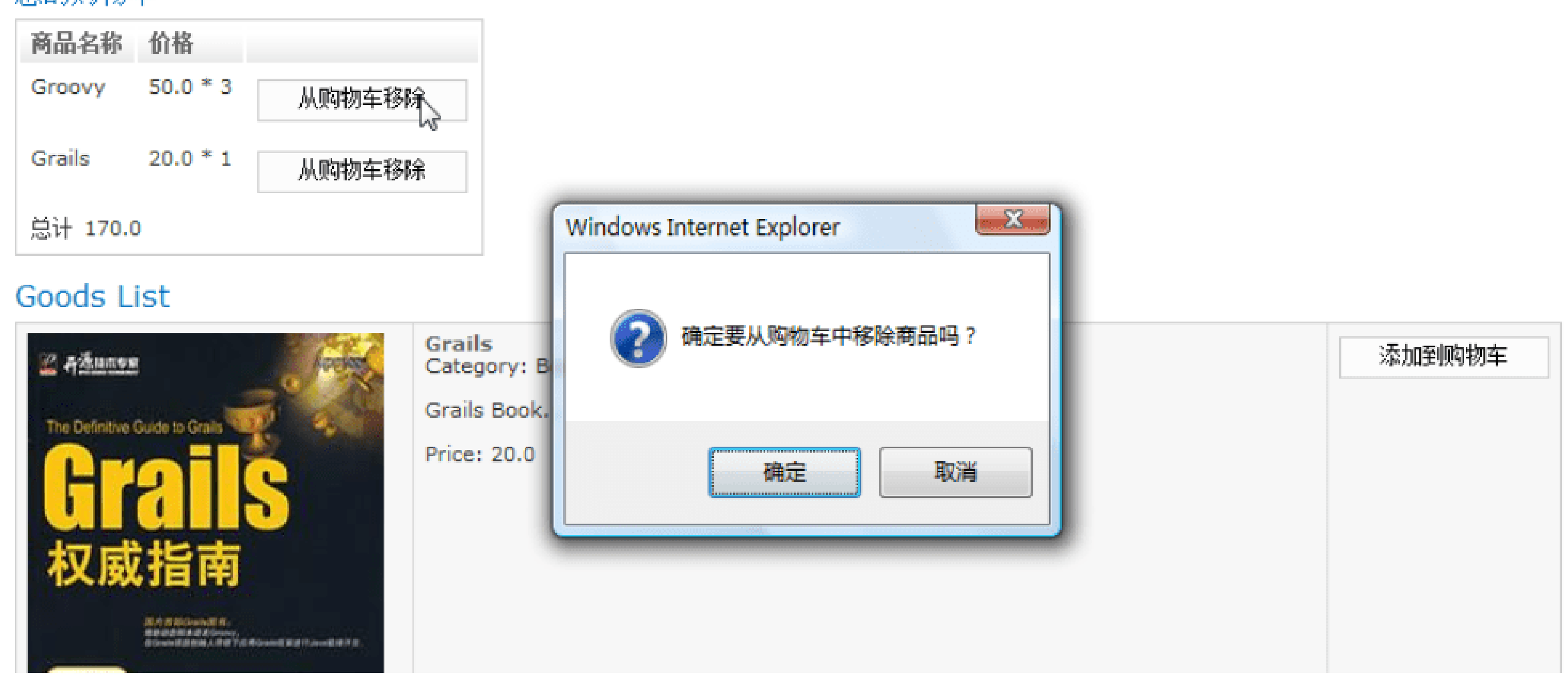


图 7-2 单击“从购物车移除”按钮

再单击“确定”按钮，该商品就从购物车中消失了，显示效果如图 7-3 所示。



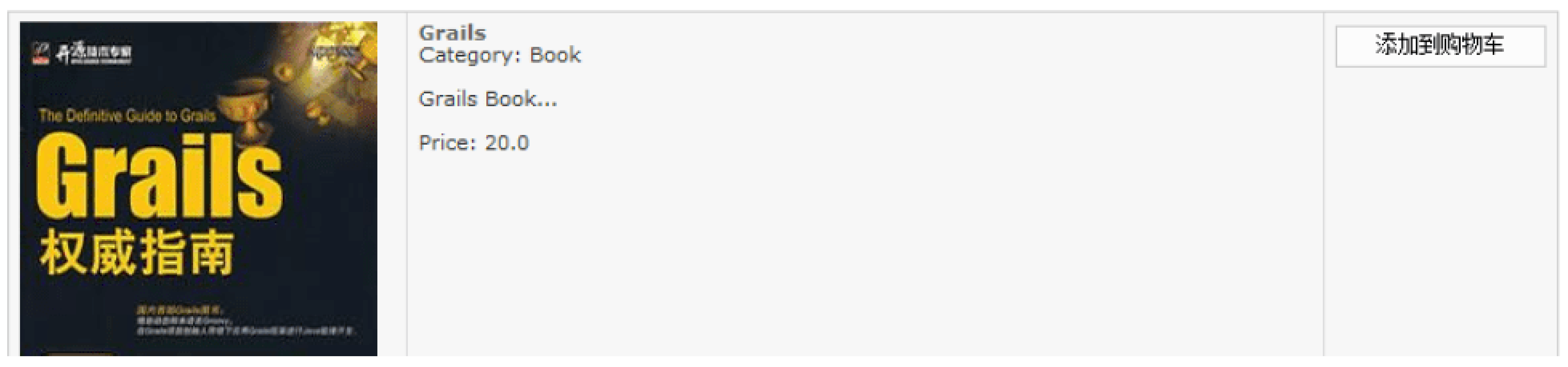


图 7-3 成功删除一条购物车中的记录

接下来，开发修改购物车中物品数量的功能，仍然在显示购物车的表格里添加一列：

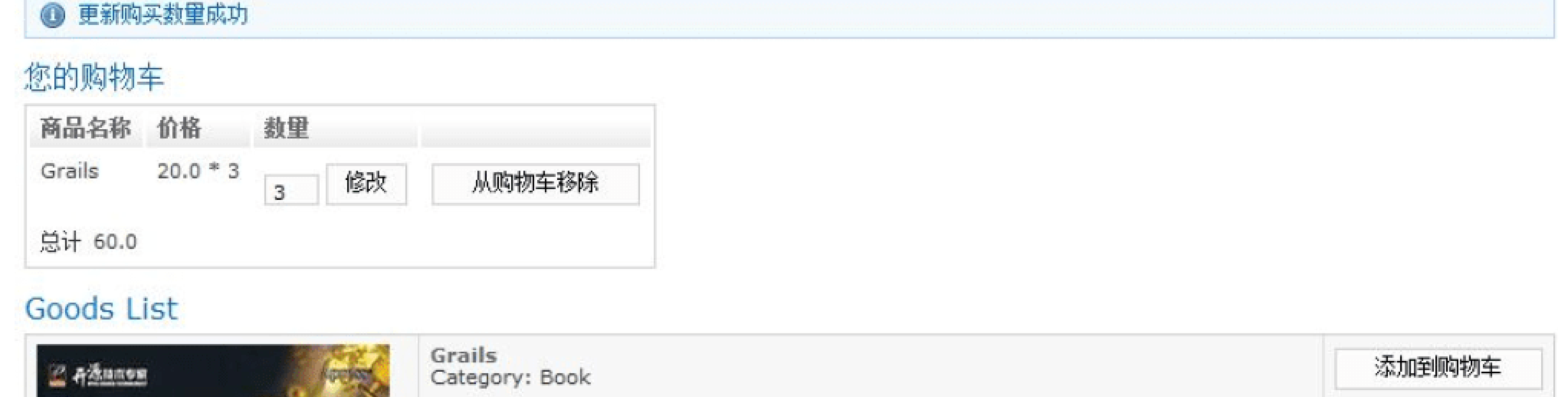
```
<td>
    <g:form action="changeItemNumber" method="post">
        <input type="hidden" name="id" value="${lineItem.id}">
        <input type="text" size="2"
            name="itemNumber" value="${lineItem.itemNumber}" >
        <input type="submit"
```

```
                value="${message (code:'cart.changeItemNumber')}" >
        </g:form>
    </td>
```

这一列包含一个文本框和一个提交按钮。表单项包含 lineItem 的 id 和要修改的商品数量。处理表单的 action 是 changeItemNumber，要实现用表单提交的数值去更新指定的 lineItem，程序代码如下：

```
def changeItemNumber = {
    def lineItem = LineItem.get(params.id)
    // 防止用户提交无效的 id
    if(lineItem && lineItem.cart.id == session.cartId) {
        lineItem.properties = params
        if(!lineItem.hasErrors() && lineItem.save() ){
            flash.message=
            message(code:'lineItem.itemNumber.update.success')
        } else {
            flash.message=
            message(code:'lineItem.itemNumber.update.falied')
        }
    }
    redirect(action:list)
}
```

这部分程序与 Grails 自动生成的 update action 十分相似，这里不再赘述。再次刷新页面，单击“修改”按钮，可以看到运行效果如图 7-4 所示。



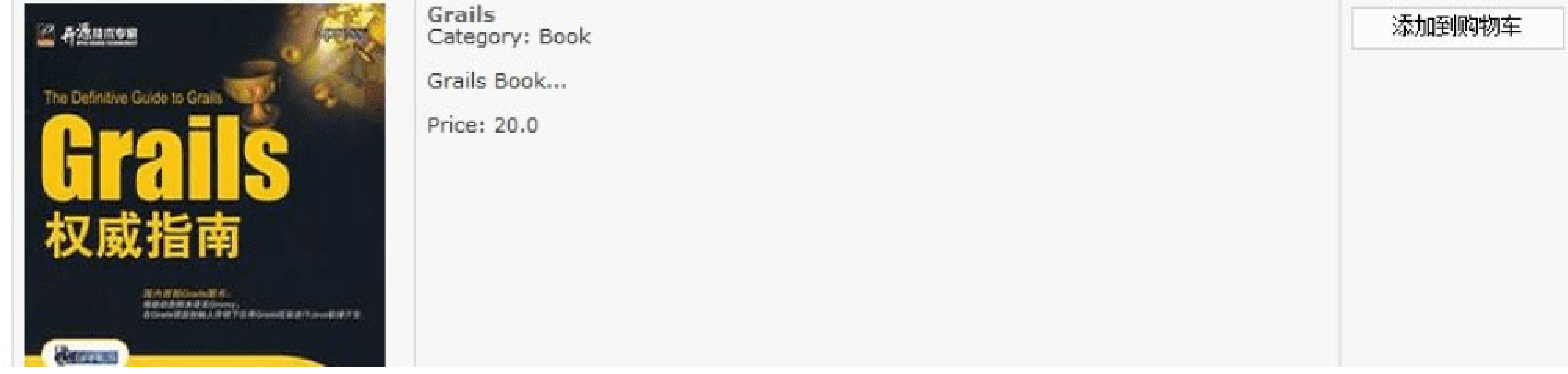


图 7-4　修改购买数量

本节实现了购物车的显示，实现了向购物车添加商品，还实现了从购物车中移除商品和编辑购物车中商品的数量。下一节将开发提交订单的功能。

7.2 订单的提交

7.2.1 定义订单的 Domain 类

订单的内容肯定要包含用户、用户购买的商品信息。将购物车包含进来，就包含了全部的已购商品信息。订单中应该还包含一份单独的收货人信息，因为提交订单的用户可能会不使用用注册时提交的地址作为收货地址。设计订单的 Domain 类如下：

```
enum OrderStatus {NEW,SHIPPED,CLOSE}
class Orders {
    Cart cart                   //订单对应的购物车
    String receiverName         //收货人姓名
    String address              //收货人地址
    String phone                //收货人电话
    BigDecimal price            //订单总价
    OrderStatus status          //订单状态
    Date orderDate              //下单时间
    Date shipDate               //发货时间
    static belongsTo = [user:User]
    static constraints = {
        receiverName(size:2..10,blank:false)
        phone(matches:/\d{7,11}/,blank:false)
        address(maxSize:200,blank:false)
        price(min:0.0)
        shipDate(nullable:true)
        cart(validator: { cart ->
            if(cart && LineItem.countByCart(cart) > 0) {
                //countBy 与 findBy 类似，用于求 count
                return true
            }
            return 'blank'
        })
    }
}
```

由于 order 是 SQL 语句中的保留字，因此用“Orders”作为 Domain 的类名。枚举 OrderStatus 中为订单定义了 3 种状态：NEW（新订单）、SHIPPED（已发货）、CLOSE（处理完毕）。orderDate 和 shipDate 两个属性分别记录了下单的时间和发货的时间。由于未发货的订单没有发货时间，因此在 constraints 中允许 shipDate 的值可以为 null。Orders 类中还对 cart 定义了一个 validator，用于防止用户提交不包含任何商品的订单。

7.2.2 提交订单的表单页面

接下来，运行命令 grails generate-all Orders，让 Grails 自动生成提交订单的 Scaffold

页面。

这里仍选用 create 页面作为订单提交的页面，不过这回 Grails 生成的页面就离预期相差得太远了。但是，如果能够在这个基础上进行修改，也明显减轻了负担。

接下来修改 create 页面。首先，要将购物车的内容显示在页面上，然后将提交订单时不能修改的属性都去掉。

在页面上显示购物车的内容，可以将商品列表（views\goods\list.gsp）那页的部分代码粘贴过来。但这样做，又会有大量的重复代码出现。使用 Grails 提供的**模板机制**（template），可以解决这个问题。在 views 文件夹下创建一个名为_showCart.gsp 的文件，然后将 goods/list.gsp 中用于显示购物车的代码移动到_showCart.gsp 中，最后在 goods/list.gsp 中使用下面一行代码来调用模板：

```
<g:render template="/showCart" model="${[cart:cart]}" />
```

g:render 标签可以用于输出模板页面的内容。其中 template 属性为必填属性，用于指定模板页面的文件名。如果 template 的值以“/”开头，则表示模板文件在 views 文件夹下，如果不以“/”开头，则表示在当前路径下。注意/showCart 与_showCart.gsp 的对应关系。g:render 标签还可以向模板页面传递数据，通过 model 属性，可以将封装成 Map 的数据传递给模板页面，供其使用。更多关于 g:render 的内容，在后面章节里会有介绍。

通过模板机制，只需要一行代码，就可以将购物车显示在 orders\create.gsp 页面中。但此时，订单页面中购物车的部分功能无法使用。这是因为在表单中没有指定 controller，默认会提交到当前 Controller，即 OrdersController。这个问题不难解决。为_showCart.gsp 中的两个 form 都指定 controller 为“goods”即可。

但很不幸，问题并没有结束。以“移除商品”为例，removeFromCart action 的代码如下：

```
def removeFromCart = {
    def lineItem = LineItem.get(params.id)
    if(lineItem && lineItem.cart.id == session.cartId) {
        lineItem.delete()
    }
    redirect(action:list)
}
```

最后一行的跳转处又出了问题，因为如果是在提交订单的页面移除了购物车中的内容，显然应该跳转回提交订单的页面才对。同样的问题也会出现在查询时：如果在显示查询结果的页面上修改了购物车的内容，一样会跳转到 list 页面。为解决这个问题，需要向 action 提交一个参数，用于指定下一步要跳转到的 URL。为此，在_showCart.gsp 的每个表单中都加入如下的代码（goods/list.gsp 页面的 addToCart 表单中，也应该加入下面的代码）：

```
<input type="hidden" name="forwardURI"
    value="${request.forwardURI}?${request.queryString}">
```

request.forwardURI 可以获取到当前页面的 URL（不包含参数）；request.queryString 可

以获取当前 HTTP 请求的参数信息。因此通过“${request.forwardURI}?${request.queryString}”就可以得到当前请求的完整的 URL 了。经过上面的修改，得到完整的_showCart.gsp 的内容如下：

```
<div>
<g:if test="${cart.lineItems}">
<h1><g:message code="yourCart"/></h1>
<table>
    <thead>
        <th><g:message code="goods.title" /></th>
        <th><g:message code="goods.price" /></th>
        <th><g:message code="lineItem.itemNumber" /></th>
        <th></th>
    </thead>
    <tbody>
    <g:each in="${cart?.lineItems}" var="lineItem">
        <tr>
        <td>${lineItem.goods?.title}</td>
        <td>${lineItem.goods?.price} * ${lineItem.itemNumber} </td>

        <td>
            <g:form controller="goods" action="changeItemNumber"
             method="post">
                <input type="hidden" name="id" value="${lineItem.id}">
                <input type="hidden" name="forwardURI"
                value="${request.forwardURI}?${request.queryString}">
                <input type="text" size="2" name="itemNumber"
                value="${lineItem.itemNumber}" >
                <input type="submit"
                    value="${message (code:'cart.changeItemNumber')}" >
            </g:form>
        </td>
        <td>
            <g:form controller="goods" action="removeFromCart"
            method="post">
                <input type="hidden" name="id" value="${lineItem.id}">
                <input type="hidden" name="forwardURI"
                value="${request.forwardURI}?${request.queryString}">
                <input type="submit"
                value="${message (code:'cart.remove')}"
                onClick="return
            confirm('${message(code:'confirm.removeFromCart')}')">
            </g:form>
        </td>
        </tr>
```

```
        </g:each>
        </tbody>
        <tr>
            <td colspan="4">
    <g:message code="cart.totalPrice" />
    ${cart.totalPrice()}
    </td>
        </tr>
    </table>
    </g:if>
    <g:else>
        <h2><g:message code="cart.empty"/></h2>
    </g:else>
    <div>
```

相应地，还需要修改 Controller 中的程序，将 addToCart、removeFromCart、changeItemNumber 这 3 个 action 中的 redirect(action:list)替换成下面的代码：

```
if(params.forwardURI)
    redirect(url : params.forwardURI)
else
    redirect(action:list)
```

完成了显示购物车的功能，接下来开发订单提交的表单页面。首先需要删除 Grails 自动生成的表单中不合逻辑的内容：Price、Ship Date、Cart、Order Date、Status、User。然后将 Address 对应的文本框替换为多行文本框，并将 Create 按钮的标签改为“确认下单”。

然后，需要修改 OrdersController 的 create action。这里需要做两件事情：一件是读取购物车的数据，因为 create 页面上也需要显示购物车；另一件是用当前用户的 userName、phone 和 address 去初始化订单的对应属性。具体代码如下：

```
def create = {
    def orders = new Orders(params)
    def cart = orderService.prepareCart(session)
    def user = cart.user
    orders.receiverName = user.userName
    orders.address = user.address
    orders.phone = user.phone
    return ['orders':orders, cart:cart ]
}
```

提交订单的表单页面显示效果如图 7-5 所示。



图 7-5　提交订单的表单页面

显然，OrdersController 中的每个 action 都需要登录保护，此外 OrdersController 中也需要有 orderService 这个属性，使得 Spring 能够将容器中的 OrderService 实例注入进来。到目前为止，提交订单的表单页面已经基本完成。接下来要开发保存订单的功能。

7.2.3　订单的保存

订单保存，需要将 orders 记录保存到数据库，同时需要将 cart 的 status 属性设置为 CartStatus.ORDERED。这是一个比较典型的事务处理，因此很自然地就会把这部分功能放在 OrderService 类中实现。于是在 OrderService 中添加 saveOrder 方法：

```
def saveOrder(session, orders) {
    def cart = prepareCart(session)
    if(cart) {
        orders.cart = cart
        orders.user = cart.user
        orders.orderDate = new Date()
        orders.price = cart.totalPrice()
        orders.status = OrderStatus.NEW

        if(!orders.hasErrors() && orders.save() ) {
            cart.status = CartStatus.ORDERED
            if(cart.save()){
```

```
                    session.cartId = null   //由于 cart 的状态发生了变化,
                                            //所以要清除 session 中的 cartId
                    return orders           //①
                }
            }
        }
        throw new RuntimeException('Save order failed!') //②
}
```

saveOrder 方法有两个参数：一个是 session，另一个是根据表单构造的 orders 对象。订单的保存过程并无特别之处，先是为 orders 的 cart、user、orderDate、price、status 属性赋值，然后在 orders 保存成功的前提下，对 cart 进行更新并保存。只有当 orders 与 cart 的更新都成功的时候，程序才会执行到①处，返回 orders；否则，程序一定会执行到②处，抛出异常并退出。这里的抛出异常，正是为 Spring 的事务管理所准备的。当 Spring 容器检测到这个异常时，会对事务进行回滚操作，从而保证了事物操作的原子性。

OrdersController 中的 save action 只需要调用 OrderService 中的 saveOrder 方法，因此十分简单：

```
def save = {
    def orders = new Orders(params)
    try{
        orderService.saveOrder(session, orders)
        flash.message = message(code:"order.save.success")
        redirect(action:'list')
    } catch (Exception ex) { //发生异常，返回提交页面，并报告错误信息
        flash.message = message(code:"order.save.failed")
        def cart = orderService.prepareCart(session)
        render(view:'create',model:[orders:orders,cart:cart])
    }
}
```

7.3　订单的查看

完成了提交订单，接下来开发为顾客显示订单的页面，从 action 开始，list action 中需要查询当前用户的订单列表。假定需要显示 3 种类别的订单：全部订单、新订单、已经发货的订单。这 3 种状态由一个 url 的参数“status”表示：status=new 表示新订单；status=old 表示已经发货的订单；status 为其他值则表示全部显示。程序代码如下：

```
def list = {
  if(!params.max) params.max = 10
  def user = User.get(session.userId)
  if(user) {
    def orderList
```

```
        def orderCount
        params.order='desc'
        params.sort='orderDate'
        if(params.status == 'new') {
            orderList = Orders.findAllByUserAndStatus(user,
                        OrderStatus.NEW, params )
            orderCount = Orders.countByUserAndStatus(user,OrderStatus.NEW)
        } else if(params.status == 'old') {
            params.sort = 'shipDate'
            orderList = Orders.findAllByUserAndStatusNotEqual(user,
                        OrderStatus.NEW, params )
            orderCount = Orders.countByUserAndStatusNotEqual(user,
                        OrderStatus.NEW)
        } else {
            orderList = Orders.findAllByUser(user, params )
            orderCount = Orders.countByUser(user)
        }

        return [ ordersList:
            new grails.orm.PagedResultList(orderList, orderCount) ]
      }
    }
```

程序逻辑很清晰，也没有新的知识点，无非是使用 findAllBy 方法进行查询，并且用一个 map（params）去控制分页与排序。读者如果忘记了相关的知识点，可以翻看前一章回顾一下。接下来开发显示订单列表的 list.gsp 页面。

首先制作几个导航链接，分别指向“购买商品”(goods/list)、“新订单”(orders/list?status=new)、“已发货订单”(orders/list?status=old)和“全部订单”(orders/list)。然后制作显示订单列表的表格：

```
<table>
<thead>
<tr>
   <th><g:message code="orders.goods"/></th>
   <th><g:message code="orders.receiver.address"/></th>
<g:sortableColumn property="price" titleKey="orders.price"
    params="${[status:params.status]}"/>
<g:sortableColumn property="orderDate" titleKey="orders.orderDate"
        params="${[status:params.status]}"/>
<g:sortableColumn property="shipDate" titleKey="orders.shipDate"
        params="${[status:params.status]}" />
    <th></th>
</tr>
</thead>
<tbody>
```

```
<g:each in="${ordersList}" status="i" var="orders">
    <tr class="${(i % 2) == 0 ? 'odd' : 'even'}">
    <td>
<p><g:message code="orders.id" />:
${fieldValue(bean:orders, field:'id')}</p>
    <p><g:message code="orders.containsGoods" />:</p>
    <p>
    <ul>
    <g:each in="${orders.cart?.lineItems}" var="lineItem">
<li>
<g:link controller="goods" action="show"
     id="${lineItem.goods?.id}">
     ${lineItem.goods?.title}
</g:link>
    *
${lineItem.itemNumber}
</li>
    </g:each>
    </ul>
    </p>
    </td>
    <td>
        <g:message code="orders.receiver" /> :
         ${fieldValue(bean:orders, field:'receiverName')}<br/>
         <g:message code="orders.phone" /> :
         ${fieldValue(bean:orders, field:'phone')}<br/>
         <g:message code="orders.address" /> :
         ${fieldValue(bean:orders, field:'address')}
     </td>
     <td>${fieldValue(bean:orders, field:'price')}</td>
     <td><g:formatDate format="yyyy-MM-dd"
     date="${orders.orderDate}"/></td>
     <td><g:formatDate format="yyyy-MM-dd"
     date="${orders.shipDate}"/></td>
     <td>
         <g:if test="${orders.status == OrderStatus.NEW}">
             <g:message code="orders.status.new" />
         </g:if>
         <g:elseif test="${orders.status == OrderStatus.SHIPPED}">
             <g:message code="orders.status.shipped" />
             <g:form action="markAsReceived" method="post">
                <input type="hidden" name="id" value="${orders.id}"/>
                <input type="submit"
             value="${message(code:'orders.received')}"/>
             </g:form>
```

```
            </g:elseif >
            <g:elseif  test="${orders.status == OrderStatus.CLOSE}">
                <g:message code="orders.status.close" />
            </g:elseif >
        </td>
        </tr>
    </g:each>
    </tbody>
    </table>
```

标签 g:sortableColumn 会输出一个表头（<th>），并且这个表头中包含一个链接，链接中的参数可以指定排序方式。g:sortableColumn 的 property 属性决定使用什么字段进行排序。title 属性用于指定要显示的文本。如果要输出的内容需要从资源文件中读取，则可以用 titleKey 属性指定资源的 key。编写<g:sortableColumn property = "price" titleKey = "orders.price" />会输出形如 list?sort=price 的链接。如果链接中需要添加其他的参数，则需要使用 params 属性。以当前的情况为例，需要给链接的 URL 中添加一个 status 属性，因此使用了 params 属性如下：params="${[status : params.status]}"。

标签 g:formatDate 用于输出格式化的日期，format="yyyy-MM-dd"表示输出格式为“2008-08-08”。

接下来，输出分页导航：

```
<g:paginate total="${ordersList.totalCount}"
    params="${status:params.status}" />
```

页面的显示效果如图 7-6 所示。



购买商品　新订单　已发货订单　全部订单

Orders List

订单明细	收货地址信息详情	总价	下单日期 ▼	发货日期	
订单编号: 2 包含商品: • Grails * 1	收货人姓名 ：梁士兴 电话号码 ：12345678 收货地址 ：北京市，知春路，2# 401	20.00	2008-07-31		未发货
订单编号: 1 包含商品: • Groovy * 1 • Grails * 4	收货人姓名 ：梁士兴 电话号码 ：12345678 收货地址 ：北京市，知春路，2# 401	100.00	2008-07-31		未发货

图 7-6　用户订单列表

总的来说，这个页面是比较复杂的，涉及到分页、表头的排序等。Grails 通过在链接

中加入 sort、order、max、offset 这样的参数，可以让 action 中的 params 成为包含上述 key 的 Map，并通过这个 Map 影响 Domain 类对数据库的查询。这确实是一个很精巧的设计，如果读者觉得这部分内容理解起来有困难，不妨在 action 中加入一行 println params，通过反复单击页面上的排序或分页链接，查看打印出的 params 内容。通过反复多次的观察，相信一定可以理解其中的奥妙。

表格的最后一列用于显示订单状态，当状态为“已发货”时，该列还将包含一个表单，表单中有一个按钮，用户单击该按钮就表示商品已经收到。提交表单时，表单项中包含了订单的 id，相应的表单提交的 action 中的代码如下：

```
def markAsReceived = {
    def orders = Orders.get(params.id)
    if(orders && orders.user.id == session.userId
        && orders.status == OrderStatus.SHIPPED) {
        orders.status = OrderStatus.CLOSE
        orders.save()
    }
    redirect(action:'list',params:[status:'old'])
}
```

action 中执行的操作就是把指定 orders 的 status 改为 CLOSE。orders.user.id == session.userId 是为了防止恶意的提交，道理和上一节修改 lineItem 一样，这里不再赘述。

7.4　本章小结

本章实现了购物车的显示与维护、订单的提交与查看。本章的程序在功能上和逻辑上都要比前面的章节复杂不少。但无论多么复杂的功能，也都是由一些最基本的操作实现的，如数据库的增删查改、数据显示、表单提交、表单输出、表单验证、链接的制作和 Session 的读写等。Grails 自动生成的代码就是针对这些基本问题，因而它可以在一定程度上实现部分需求。对于其他更为复杂的需求，通常也可以通过修改或拼装 Grails 自动生成的代码去实现。因此，利用 Grails 自动生成的代码，可以极大地提高开发效率。修改 Grails 自动生成的代码，是使用 Grails 解决实际问题的一个基本步骤。

第8章 系统后台管理

本章将开发系统后台管理相关的内容，包括对商品和订单信息进行管理。系统后台管理需要安全级别更高的权限管理和认证机制。本章不讨论权限认证与分配相关的内容，Grails 中可以通过 Acegi、JSecurity 等插件实现相关功能，它们会在第 16.3 节进行介绍。本章的主要知识点是**页面布局的技术和文件上传**。

购物车系统的后台管理，主要包括商品管理和订单管理。后台管理的主要功能，实际上也可以通过 Grails 自动生成 CRUD 页面来实现。这里需要做的事情主要是把之前商品和订单相关的 CRUD 功能转移到后台，也就是将 GoodsController 和 OrdersContoller 中的部分 action 转移到 AdminController 下，并改成相应的命名原则，如商品的 list 方法改为 listGoods，订单的 list 方法改为 listOrders，等等。需要转移的方法可参照如下：

（1）goods.create → admin.createGoods；

（2）goods.list → admin.listGoods；

（3）goods.edit → admin.editGoods；

（4）orders.list → admin.listOrders；

（5）orders.edit → admin.editOrders。

其中，商品管理的具体实现可以参考本书第二部分第 7 章的内容，订单管理的具体实现可参考 7.3 购物车与订单的内容。

对于系统的后台管理，还需要为其单独制作一个区别于前台页面的**导航菜单**。同时，需要**实现文件上传的功能**，允许用户自由上传商品的图片。此外，对于订单管理，需要**修改订单状态**，把“未发货”修改为“已发货”。这 3 部分的内容，将分别在下面的 3 节进行介绍。

8.1 页面布局的使用

8.1.1 Grails Layout 的基础知识

到现在，相信读者已经对 Grails 有了一定了解，在它自动生成的页面基础上，可以实现大部分功能。但是，到目前为止，人们仍然会对它自动生成页面的运行效果有一些不解之处。例如，在程序的每个页面，左上角都有一个 Grails 的 logo，如图 8-1 所示。

但事实上，在 views 的普通 GSP 文件中，看不到输出 logo 图片的 HTML 标签。之所以会看到 logo，是因为 Grails 使用了一种页面布局（layout）的机制，并且提供了一个包含

Grails logo 的默认 layout 页面。Layout 机制使得不同的页面能够使用相同的 layout，从而在多个页面中得到一致或相似的页面风格。显然，使用 layout 技术，可以实现对 UI 代码的重用，从而提高开发效率和程序质量。



图 8-1　Grails 的 logo

Grails 的 layout 技术是基于成熟可靠的 SiteMesh 框架实现的。按照 SiteMesh 官方的说法："SiteMesh 是一个网页布局装饰框架和网页应用集成框架，能够为有多个页面的大型网站构建一致的外观和效果，相同的导航和布局"。这段话理解起来可能有些抽象，引用官方网站的一个图片，可以很好地解释它，如图 8-2 所示。

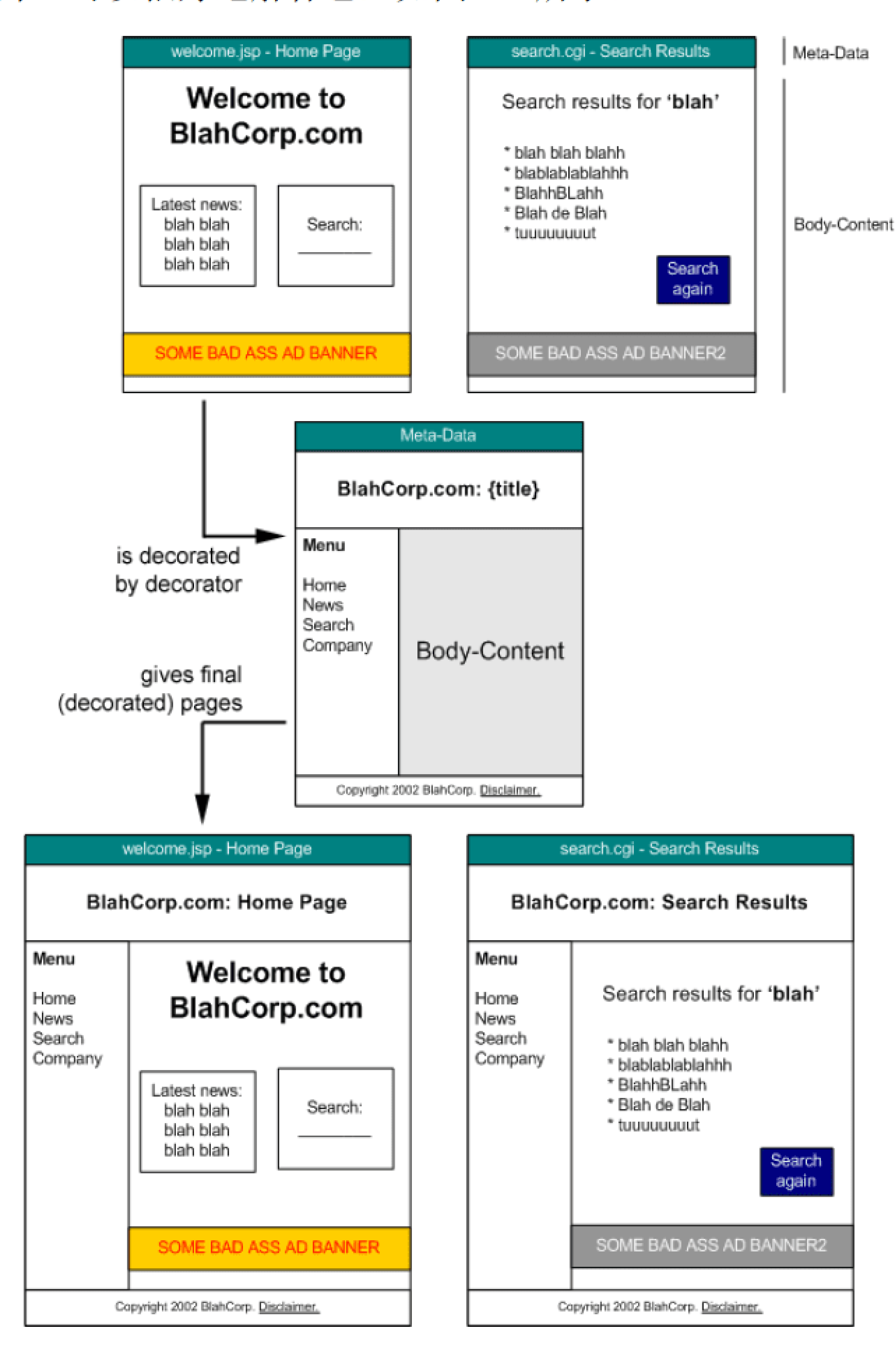


图 8-2　SiteMesh 的作用

图 8-2 可以分为 3 部分，最上面的两个页面是原始页面，其中一个页面是 JSP，另一个是 CGI 脚本；中间是一个装饰器（decorator），这是 SiteMesh 的核心；最下面的两个页面是最终页面，它的显示内容是用 decorator 装饰过的原始页面，拥有一致的布局风格。Grails 的 layout 使用 SiteMesh 框架，通过定制 layout，使得原始的 GSP 页面在经过 decorator 装饰之后，对外输出具有统一风格的页面。

理解了 SiteMesh 的工作原理之后，将继续介绍如何在 Grails 里来使用基于 SiteMesh 框架的 layout 技术。

1. 创建 layouts

在 Grails 的目录结构里，decorator 默认放置在 grails-app\views\layouts 目录下。可以先看一个典型的 decorator 页面：

```
<html>
    <head>
    <title><g:layoutTitle default="Grails" /></title>
    <link rel="stylesheet"
    href="${createLinkTo(dir:'css',file:'main.css')}" />
    <link rel="shortcut icon"
    href="${createLinkTo(dir:'images',file:'favicon.ico')}"
    type="image/x-icon" />
    <g:layoutHead />
    <g:javascript library="application" />
    </head>
    <body>
        <div id="spinner" class="spinner" style="display:none;">
          <img src="${createLinkTo(dir:'images',file:'spinner.gif')}"
        alt="Spinner" />
        </div>
        <div class="logo">
        <img src="${createLinkTo(dir:'images',file:'grails_logo.jpg')}"
        alt="Grails" />
        </div>
       <g:layoutBody />
   </body>
</html>
```

从上面的代码可以看出，decorator 页面中使用了几个关键的 Grails 标签（tag）：layoutHead、layoutTitle 和 layoutBody。它们的功能分别为：

（1）layoutHead——向目标页面输出原始页面文档头（head）的内容；

（2）layoutTitle——向目标页面输出原始页面标题（title）的内容；

（3）layoutBody——向目标页面输出原始页面文档主体（body）的内容。

接下来继续讨论在 Grails 里原始页面如何与 decorator 页面进行交互。

2. 在原始页面中调用 decorator

Grails 中提供了简便的调用 decorator 的方式，只需在原始页面里添加 meta 标签，如下面的例子所示：

```
<html>
<head>
    <title>An Example Page</title>
    <meta name="layout" content="main"></meta>
</head>
<body>This is my content!</body>
</html>
```

通过使用 name 值为 layout 的 meta 标签，就可以调用 decorator。其中 content 属性的值用于指定 decorator 的文件名（main 对应 decorator 页面的文件名是 grails-app\views\layouts\main.gsp）。在使用了上述的 meta 标签后，可以查看输出页面的 HTML 代码如下所示：

```
<html>
   <head>
      <title>An Example Page</title>
      <link rel="stylesheet" href="/GDepot/css/main.css" />
      <link rel="shortcut icon" href="/GDepot/images/favicon.ico"
    type="image/x-icon" />
      <meta name="layout" content="main"></meta>
      <script type="text/javascript"
   src="/GDepot/js/application.js"></script>
   </head>
   <body>
      <div id="spinner" class="spinner" style="display:none;">
         <img src="/GDepot/images/spinner.gif" alt="Spinner" />
      </div>
      <div class="logo"><img src="/GDepot/images/grails logo.jpg"
   alt="Grails" /></div>
      This is my content!
   </body>
</html>
```

8.1.2 为系统后台管理创建统一的 decorator

首先，创建带左侧导航的 decorator 页面（grails-app\views\layouts\admin.gsp）：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
```

```
    <head>
        …
    </head>
    <body>
      <div id="spinner" class="spinner" style="display:none;">
       <img src="${createLinkTo(dir:'images',file:'spinner.gif')}"
       alt="Spinner" />
      </div>
  <div class="logo">
  <img src="${createLinkTo(dir:'images',file:'grails_logo.jpg')}"
       alt="Grails" />
  </div>
      <div id="mainBody" >
        <div id="leftMenu" >
          <ul>
              <li>
                  <h1>
                      <g:link controller="admin" action="listGoods">
                        <g:message code="admin.link.listGoods"/>
                     </g:link>
                  </h1>
             </li>
          <li>
             <h2>
                 <g:link controller="admin" action="createGoods">
                   <g:message code="admin.link.createGoods"/>
               </g:link>
            </h2>
          </li>
           <li>
              <h1>
                <g:link controller="admin" action="listOrders">
                  <g:message code="admin.link.listOrders"/>
                </g:link>
             </h1>
           </li>
           …
         </ul>
       </div>
       <div id="mainContent">
         <g:layoutBody />
       </div>
      </div>
    </body>
  </html>
```

斜体部分的代码 leftMenu div，创建了左侧的菜单栏。如果原始页面调用了 admin.gsp，则实际显示效果如图 8-3 所示。

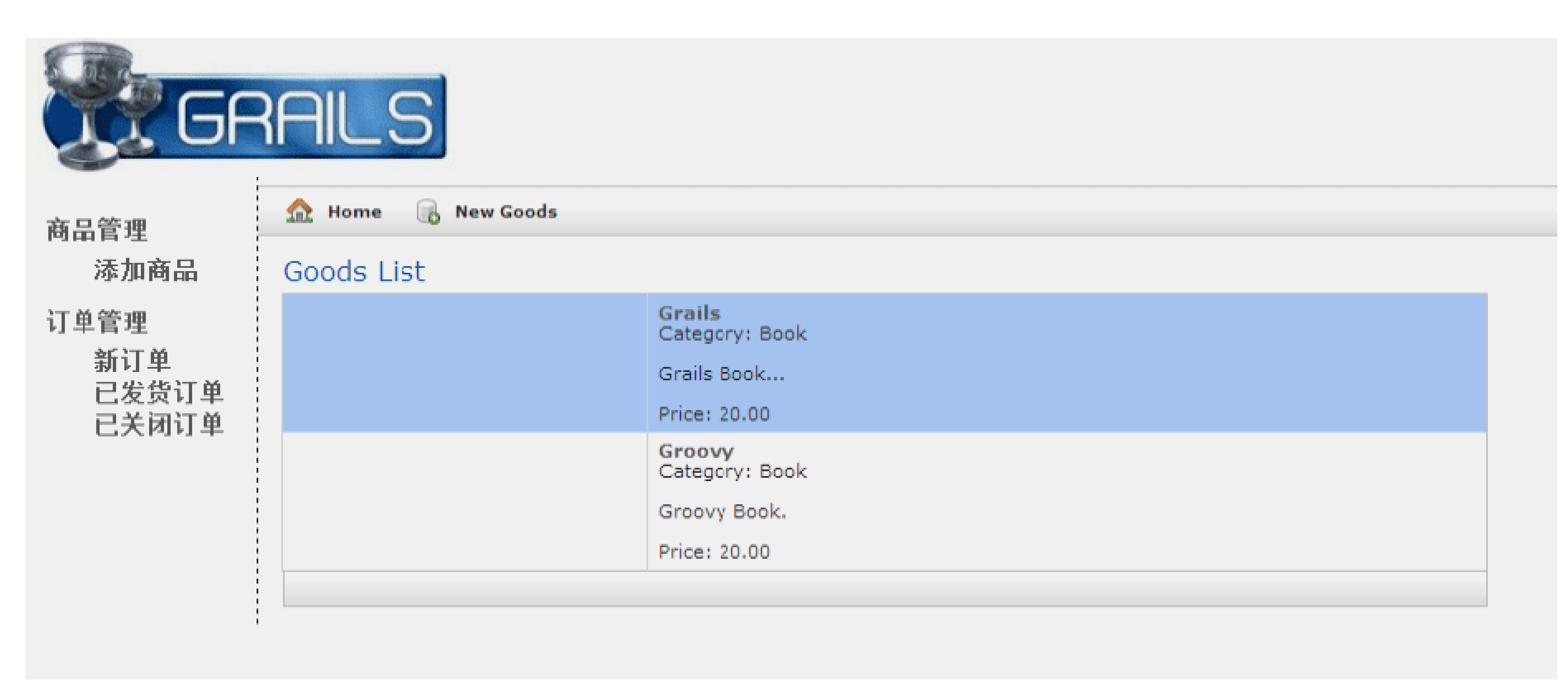


图 8-3　经 admin.gsp 装饰后页面含有左侧菜单和 logo

在实际的应用中，还经常会遇到这样的情况，页面的整体布局风格是一致的，但不同的栏目有不同的二级导航菜单。以实际最常见的两栏结构为例，若二级菜单栏有 4 种，如图 8-4 所示。

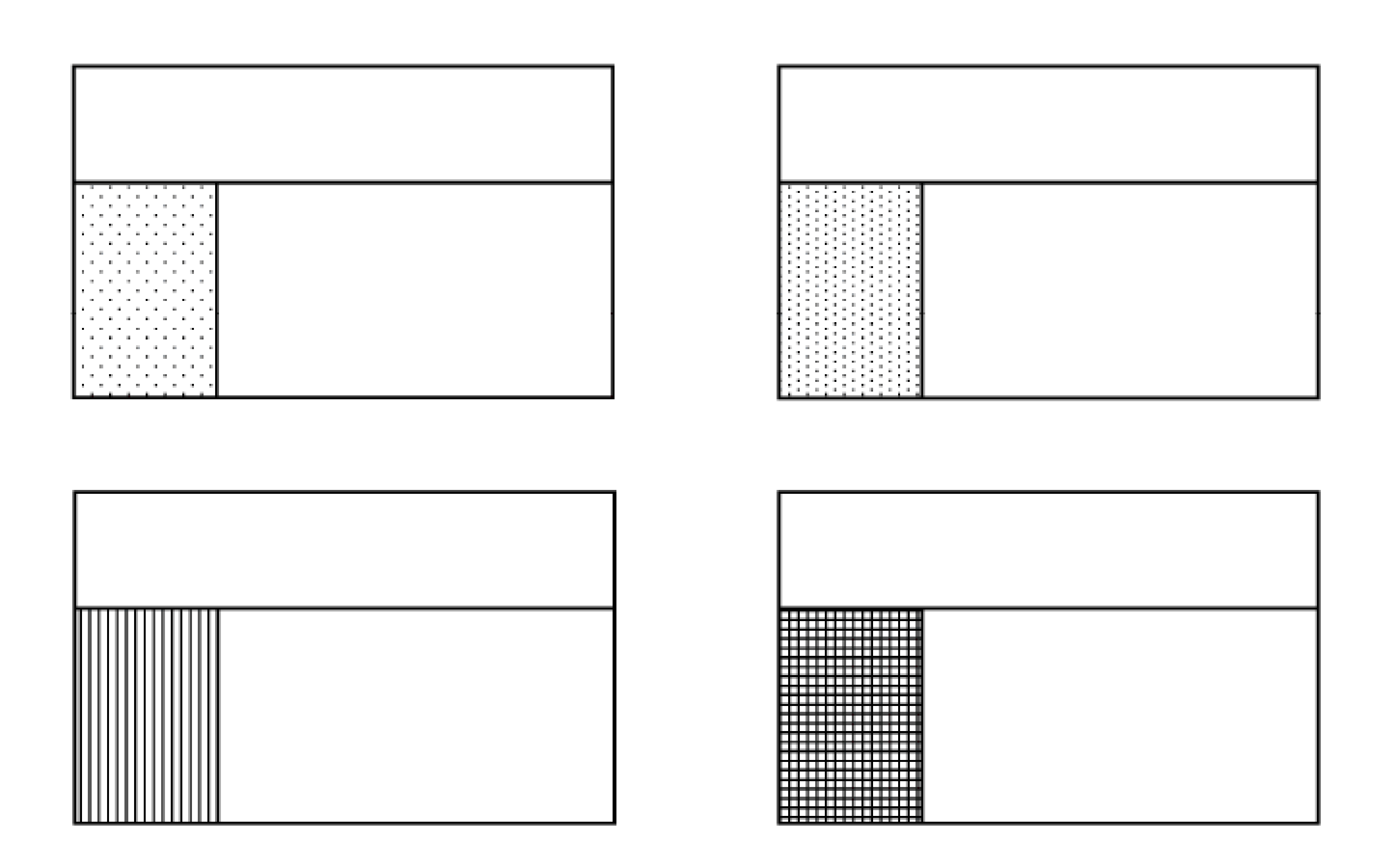


图 8-4　不同模块使用不同的 leftMenu，但整体结构相似

如果对每种不同的二级导航菜单做一个 decorator，那就需要创建 4 个相似的 decorator。如果这样做，会出现大量的重复代码，也完全违背了使用 layout 技术的初衷。

针对这样的需求，应该考虑将 menu 和 decorator 解耦合。具体作法是：将 menu 内容提取出来，创建成独立的 template 页面，然后在 decorator 页面中用 render 标签对其内容进行输出（回忆一下购物车页面的显示，也是用 render 标签输出模板）：

```
<div id="leftMenu" >
    <g:render template="adminMenu" />
</div>
```

创建模板\grails-app\views\admin_adminMenu.gsp（再次注意 template 模板文件的命名方式，名字需以下划线开始），其内容为：

```
<%@ page contentType="text/html;charset=UTF-8" %>
<ul>
<li>
<h1>
<g:link controller="admin" action="listGoods">
<g:message code="admin.link.listGoods"/>
</g:link>
</h1>
</li>
<li>
<h2>
<g:link controller="admin" action="createGoods">
<g:message code="admin.link.createGoods"/>
</g:link>
</h2>
</li>
<li>
<h1>
<g:link controller="admin" action="listOrders">
<g:message code="admin.link.listOrders"/></g:link></h1></li>
<li>
<h2>
<g:link controller="admin" action="listOrders">
<g:message code="orders.new"/>
</g:link>
</h2>
</li>
    …
</ul>
```

使用 render 模板的做法，实现了 decorator 与实际内容的解耦合，但这么做还没有完全解决使用同一 decorator 显示不同 leftMenu 的实际需求。还需要实现由原始页向 decorator 页面传递具体的 leftMenu 模板的名称。

Grails 中封装了 SiteMesh 的 pageProperty 功能，被装饰页面（原始页）可以向 decorator 传递参数。而 decorator 页面可以通过 pageProperty 标签读取内容。因而，在原始页面中可以通过参数向 decorator（admin.gsp）传入实际的 leftMenu 模板的名称。

对于上面的例子，需要在原始页中加入：

```
<meta name="menu" content="adminMenu">
```

其中，name 属性决定了参数名，而 content 属性决定了参数的值。

在 decorator（admin.gsp）中，可以使用下面的代码读取参数的内容（注意参数名是

meta.menu）：

```
<g:pageProperty name="meta.menu" default="defaultMenu"/>
```

或

```
${pageProperty (name:'meta.menu', default:'defaultMenu')}
```

相应地，要根据参数的内容输出模板页面，则应写成：

```
<g:render
        template="${pageProperty (name:'meta.menu', default:
        'defaultMenu')}"    />
```

8.2　文件上传的实现

8.2.1　开发表单页面

在 Web 应用中，常会遇到文件上传的需求。例如，在购物车的应用中，管理员用户需要为商品上传相应的图片。本节将讨论如何在 Grails 中实现文件上传。

在处理上传文件前，需要先创建上传文件的表单页面。打开编辑商品的 GSP 页面 grails-app/views/admin/editGoods.gsp，需要设置如下代码，这里仅摘录表单相关的内容：

```
<g:form method="post" action="updateGoods" enctype="multipart/form-data">
    <input type="hidden" name="id" value="${goods?.id}" />
<g:select optionKey="id" optionValue="categoryName"
    from="${Category.list()}" name="category.id"
    value="${goods?.category?.id}" >
</g:select>
    <input type="text" id="title" name="title"
value="${fieldValue(bean:goods,field:'title')}"/>
    <textarea id="description" name="description">
    ${fieldValue(bean:goods,field:'description')}
    </textarea>
    <input type="file" name="savedFileName" />
    <input type="text" id="price" name="price"
    value="${fieldValue(bean:goods,field:'price')}" />
    <input type="submit" value="save"/>
</g:form>
```

关于这段代码，需要注意两个地方：

（1）enctype 的属性设置为"multipart/form-data"；

（2）input 的 type 属性设置为 file。

表单页面的效果如图 8-5 所示。

图 8-5　编辑商品的表单（含文件上传）

8.2.2　在 Controller 中接收文件

在 web-app 路径下创建名字为“file”的文件夹，该文件夹将用于保存用户上传的文件。对于用户提交的文件，通常不能直接使用原始名称保存在服务器的硬盘上，因为这样将很容易引起重名的冲突。常见的解决办法是用随机产生的字符串对其重新命名（通常还要保留原来的扩展名）。

接下来，实现接收上述表单的 updateGoods action：

```
def updateGoods = {
def goods = Goods.get( params.id )
def f = request.getFile("savedFileName")
def fileName = f.originalFilename
def savedFileName = new Random().nextInt(1000000) +
    "${fileName[ fileName.lastIndexOf('.') .. -1 ]}"

    if(!f.empty) {
        f.transferTo(new File(servletContext.getRealPath("/file/") +
            savedFileName))
    }
    else {
    flash.message = 'file cannot be empty'
```

```
        render(view:'edit')
        }

        if(goods) {
            goods.properties = params
            goods.photoUrl = createLinkTo(dir:"file", file:savedFileName)
            if(!goods.hasErrors() && goods.save()) {
                flash.message = "Goods ${params.id} updated"
                redirect(action:listGoods,id:goods.id)
            }
            else {
                render(view:'edit',model:[goods:goods])
            }
        }
        else {
            flash.message = "Goods not found with id ${params.id}"
            redirect(action:editGoods,id:params.id)
        }
    }
```

上面的 updateGoods 操作会从表单请求中读取上传文件。通过 **def f = request.getFile("savedFileName")**，可读取上传的文件。通过 originalFilename 属性可以获取用户提交的文件名。在服务器端保存的文件命名规则为“随机数+原始文件扩展名”，然后根据该规则，使用 **transferTo** 方法将所上传的文件存储到服务器 web-app\file 下。

在这里，使用 **servletContext.getRealPath** 得到 web-app 文件夹在硬盘上的实际存储路径。

对于 file 文件夹中的文件，可以使用 **createLinkTo** 取出在浏览器上访问该文件的 URL，最后将 goods 保存到数据库中。createLinkTo 是 GSP 的一个标签，具体的使用方法可以参考第 14 章。

8.3 修改订单状态

订单的管理与 7.3 节介绍的订单查看比较相似，不同的是本节提供将订单的状态从“新订单”改为“已发货”。

首先，为订单的状态制作一个按钮链接，使其可以通过单击该按钮修改状态。在 editOrders.gsp 下添加如下代码：

```
<td>
    <g:if test="${orders.status == OrderStatus.NEW }">
        <g:message code="orders.status.new" />
        <g:form action="markAsShipped" method="post">
            <input type="hidden" name="id" value="${orders.id}"/>
```

```
                <input type="submit"
                value="${message(code:'orders.shipped)}"/>
            </g:form>
        </g:if >
    </td>
```


然后，在 AdminController 里添加 markAsShipped 方法，把指定 orders 的 status 改为 SHIPPED：

```
def markAsShipped = {
    def orders = Orders.get(params.id)
    if(orders && orders.status == OrderStatus.NEW) {
        orders.status = OrderStatus.SHIPPED
        orders.save()
    }
    redirect(action:'listOrders',params:[status:'old'])
}
```

实现修改订单任务，无需新的知识点，因此没有做详细的介绍，读者可以参考 7.3 节的实现过程。

8.4　本章小结

本章主要实现了系统的后台管理，包含的新知识点有：如何使用 layout 技术实现页面代码的重用，以及如何在 Grails 中进行文件上传。到目前为止，购物车应用的基本开发暂时告一段落了。接下来将讨论 Grails 的自动化测试技术。

第9章 Grails 的自动化测试

本书的第二部分设计并实现了一个购物车的例子，采取了一种增量式的开发方法。每一次增量实现了一个或多个最终用户功能，系统在“渐近”的变化过程中不断完善。使用这样的开发方法，可以很好地与客户交流，可以及时地进行修改和调整。但是，这样的开发过程，还算不上是真正的敏捷开发，原因是测试没有跟上。如果没有自动化测试程序作保证，每一处修改都有可能给系统带来 bug。因而，从这一点上来讲，自动化测试是进行敏捷开发的前提。

Grails 对自动化测试框架进行了很好的集成。本章将介绍如何利用 Grails 集成的自动化测试框架进行测试。

9.1 Grails 自动化测试基础知识

Grails 的测试是基于 Groovy Test 框架的，而 Groovy Test 主要是基于 JUnit 的，熟悉 JUnit 的读者应该不会对 Grails 的测试感到陌生。回顾第二章，Grails 创建的项目默认将自动化测试程序（test case）放到 test 文件夹。而 test 文件夹中又包含两个子文件夹，分别是 unit 和 integration。放在 integration 文件夹中的 test case 在运行时需要 Grails 框架“参与”，因而通常用于测试需要 Grails 框架参与的场景，如数据库的查询与更新等。放在 unit 文件夹中的 test case 在运行时不需要 Grails 框架的参与，因而通常用于对普通的 Java 对象、Groovy 对象进行测试。

在使用 Grails 的命令创建 Domain 和 Controller 的时候，Grails 会主动创建相应的 test case。当然，也可以使用 grails create-integration-test 命令或 grails create-unit-test 去创建 test case。所有的 test case 都是 GroovyTestCase 类的子类，表现形式如下：

```
class UserTests extends GroovyTestCase {
    void testSomething() {

    }
}
```

可以在 TestCase 中编写多个名为 testXXX 的方法，每一个 testXXX 方法都相当于一个测试点，都会被 Junit 测试框架调用并执行测试。GroovyTestCase 类是 junit.framework.TestCase 的子类，包含了大量的断言方法，通过这些断言，可以对程序的

运行结果进行验证，从而实现自动化测试，例如：

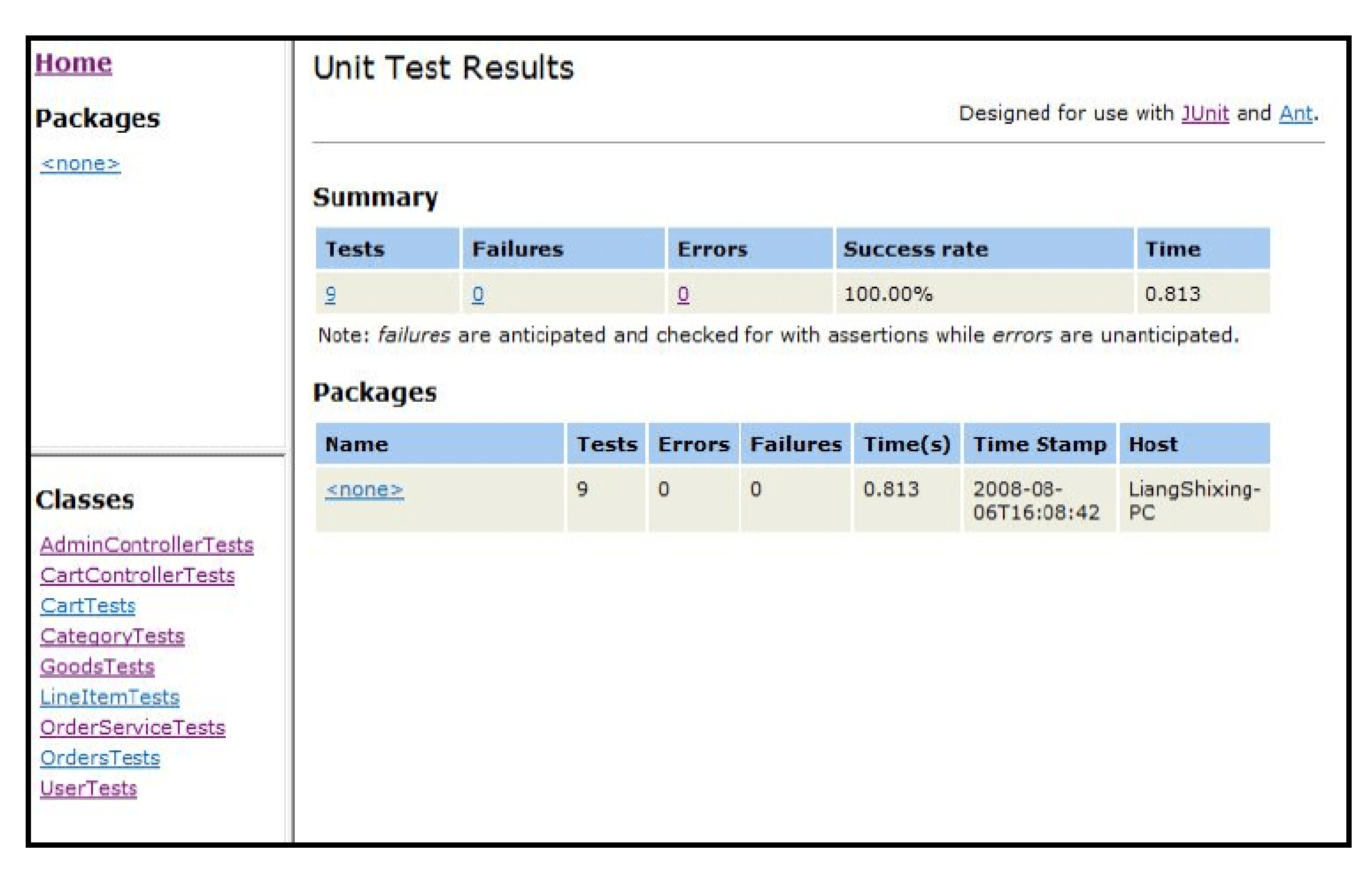
```
void testUserVerify() {
    def user= new User(userName:'aabbcc')
    assertNull(user.save())
    assertTrue(user.hasErrors())
}
```

上面的例子使用了 Grails 框架提供的 Domain 方法，因而应该放在 integration 文件夹中。它对 User 类的数据验证进行了测试：因为 User 类的属性都不能为空，所以它的保存不能成功，并且断言 save 方法会返回 null；同理，断言 hasErrors 方法返回 true。当然，也可以直接使用 groovy 的 assert 方法进行断言，具体使用哪个，可以根据自己的习惯进行选择。

Grails 中使用 grails test-app 命令可以执行测试用例。如果不指定测试用例的名称，则对全部测试用例进行测试；如果指定名称，则可以仅执行某一个测试用例。测试用例是在 test 环境下进行的（回忆一下前面配置的 DataSource，一共有 3 种环境：development、test 和 production），运行**每一个**测试用例之前，都会先清空测试环境的数据库。运行 grails test-app，可能会看到如下的输出：

```
...
Running 9 Integration Tests...
Running test AdminControllerTests...
                 testSomething...SUCCESS
Running test CartControllerTests...
                 testSomething...SUCCESS
Running test CartTests...
                 testSomething...SUCCESS
Running test CategoryTests...
                 testSomething...SUCCESS
Running test GoodsTests...
                 testSomething...SUCCESS
Running test LineItemTests...
                 testSomething...SUCCESS
Running test OrderServiceTests...
                 testSomething...SUCCESS
Running test OrdersTests...
                 testSomething...SUCCESS
Running test UserTests...
                 testUserVerify...SUCCESS
Integration Tests Completed in 1127ms
-------------------------------------------------------
...
```

同时，Grails 还会生成一份测试结果的报告，存放在 test\reports 文件夹下面。打开 HTML 格式的报告文件（test/reports/html/index.html），可以看到非常详细的测试结果，如图 9-1 所示。



Home

Packages

<none>

Classes

AdminControllerTests
CartControllerTests
CartTests
CategoryTests
GoodsTests
LineItemTests
OrderServiceTests
OrdersTests
UserTests

Unit Test Results

Designed for use with JUnit and Ant.

Summary

Tests	Failures	Errors	Success rate	Time
9	0	0	100.00%	0.813

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Packages

Name	Tests	Errors	Failures	Time(s)	Time Stamp	Host
<none>	9	0	0	0.813	2008-08-06T16:08:42	LiangShixing-PC

图 9-1　执行测试后生成的报告

9.2　编写测试用例

使用 Grails 可以对 Domain、Service、Controller、Taglib（标签库）进行测试，接下来就逐个举例讲解。

9.2.1　对 Domain 类进行测试

由于 Domain 类中对数据库操作的方法是 Grails 所提供的，即使出现问题，普通用户也难以解决。因此，对 Domain 类的测试重点应集中在对自己定义的方法和自己定义的 validator 的测试。就本书购物车应用的几个 Domain 类而言，主要应对 Cart 类的 totalPrice 方法，和 Orders 类的自定义 validator 进行测试。相应地，对 Cart 的测试代码如下：

```
class CartTests extends GroovyTestCase {
    private def cartId = null
    private def totalPrice = null
    void setUp() {
        def user = new User(userName:'liang' ,
                password: 'aabbcc'.encodeAsPassword(),
                email:'xyz@abc.com' ,
                phone: '13810000000' ,
                address: "bejing china" )
        assertNotNull(user.save(flash:true))

        def category = new Category(categoryName:'book')
        assertNotNull(category.save(flash:true))

        def book1 = new Goods(title:'book1', price:20.5 ,
```

```
            description:'', photoUrl:'', category: category )
        assertNotNull(book1.save(flash:true))

        def book2 = new Goods(title:'book1', price:10.5 ,
            description:'', photoUrl:'', category: category )
        assertNotNull(book2.save(flash:true))

        def cart = new Cart(status:CartStatus.NEW, user:user )
        assertNotNull(cart.save(flash:true))

        def lineItem1 = new LineItem(cart:cart, goods:book1,
        itemNumber: 2)
        assertNotNull(lineItem1.save(flash:true))

        def lineItem2 = new LineItem(cart:cart, goods:book2,
        itemNumber: 3)
        assertNotNull(lineItem2.save(flash:true))

        cart.refresh()
        cartId = cart.id
        totalPrice = book1.price * lineItem1.itemNumber
                    + book2.price * lineItem2.itemNumber
    }

    void testTotalPrice() {
        println 'test for total price'
        def cart = Cart.get(cartId)
        println cart.totalPrice()
        assertEquals(totalPrice , cart.totalPrice())

    }
}
```

与标准的 Junit TestCase 类似，首先在 setUp 方法中写一些初始化的代码。这里创建了相关 User、Cart、LineItem、Goods、Category 的记录。然后在 testTotalPrice 方法中，断言 cart.totalPrice()的返回值要和已知的购物车中商品价格总和相等。

在这个 test case 中，不需要编写 tearDown()方法删除数据。因为 Grails 会自动清空 test case 中编写的保存到数据库中的数据[1]，这样就不用考虑残留测试数据对下一次测试带来的影响，从而可以极大地方便编写和执行 test case。

接下来，对 Orders 类的自定义 validator 进行测试。

[1] Grails 进行 integration test 时，会自动在每个 test case 执行完成后，提交到数据库 transaction 回滚，从而避免了在测试时产生残留数据。

```
class OrdersTests extends GroovyTestCase {
    private def noneEmptyCartId = null
    private def emptyCartId = null

    void setUp() {
        def user = new User(userName:'liang' ,
                password: 'aabbcc'.encodeAsPassword(),
                email:'xyz@abc.com' ,
                phone: '13810000000' ,
                address: "bejing china" )
        assertNotNull(user.save(flash:true))

        def category = new Category(categoryName:'book')
        assertNotNull(category.save(flash:true))

        def book1 = new Goods(title:'book1', price:20.5 ,
            description:'', photoUrl:'', category: category )
        assertNotNull(book1.save(flash:true))

        def book2 = new Goods(title:'book1', price:10.5 ,
            description:'', photoUrl:'', category: category )
        assertNotNull(book2.save(flash:true))

        def cart = new Cart(status:CartStatus.NEW, user:user )
        assertNotNull(cart.save(flash:true))

        def lineItem1 = new LineItem(cart:cart, goods:book1,itemNumber: 2)
        assertNotNull(lineItem1.save(flash:true))

        def lineItem2 = new LineItem(cart:cart, goods:book2,itemNumber: 3)
        assertNotNull(lineItem2.save(flash:true))

        cart.refresh()
        noneEmptyCartId = cart.id

        cart = new Cart(status:CartStatus.NEW, user:user )
        assertNotNull(cart.save(flash:true))
        cart.refresh()
        emptyCartId = cart.id
    }
    void testOrdersVerify() {
        def orders = new Orders(cart:Cart.get(emptyCartId))
        orders.validate()
        assertTrue(orders.errors.hasFieldErrors('cart'))
```

```
            orders = new Orders(cart:Cart.get(noneEmptyCartId))
            orders.validate()
            assertFalse(orders.errors.hasFieldErrors('cart'))
        }
    }
```


OrdersTests 也是先在 setUp 方法中创建了一些初始数据，不同的是，这一次准备了两个购物车，一个包含两件商品，另一个不包含商品。然后在 testOrdersVerify 方法中创建两个 orders，分别对应两个购物车，调用 validate 方法，执行数据验证。此时，引用空购物车的 orders 应该在 cart 字段上存在错误；同时，另一个 orders 在 cart 字段上没有错误。

9.2.2 对 Service 类进行测试

在 Grails 的 integration 测试中，Service 类的实例也可以通过 Spring 的 IoC 容器注入到 test case 中，因此可以很轻松地得到容器中的实例，从而测试变得更加容易：

```
    class OrderServiceTests extends GroovyTestCase {
        def orderService //这个 orderService 也不需要进行初始化，
                    // Spring 会将 orderService 的实例注入进来
        void testSomething() {

        }
    }
```

接下来开发用于测试保存订单的 test case。这个用例在执行失败时会抛出异常，这里可以使用 shouldFail 方法来进行断言。shouldFail 方法要求传入一个闭包，若闭包在执行有异常抛出时，则测试成功，否则测试失败。TestCase 和程序代码如下：

```
    class OrderServiceTests extends GroovyTestCase {
        def orderService                  //由 Spring 容器注入 orderService
        private def cartId = null
        private def userId = null

        void setUp() {
            def user = new User(userName:'liang' ,
                    password: 'aabbcc'.encodeAsPassword(),
                    email:'xyz@abc.com' ,
                    phone: '13810000000' ,
                    address: "bejing china" )
            assertNotNull(user.save(flash:true))

            user.refresh()
            userId = user.id

            def category = new Category(categoryName:'book')
```

```
        assertNotNull(category.save(flash:true))


        def book1 = new Goods(title:'book1', price:20.5 ,
            description:'', photoUrl:'', category: category )
        assertNotNull(book1.save(flash:true))

        def book2 = new Goods(title:'book1', price:10.5 ,
            description:'', photoUrl:'', category: category )
        assertNotNull(book2.save(flash:true))

        def cart = new Cart(status:CartStatus.NEW, user:user )
        assertNotNull(cart.save(flash:true))

        def lineItem1 = new LineItem(cart:cart, goods:book1,
        itemNumber: 2)
        assertNotNull(lineItem1.save(flash:true))

        def lineItem2 = new LineItem(cart:cart, goods:book2,
        itemNumber: 3)
        assertNotNull(lineItem2.save(flash:true))

        cart.refresh()
        cartId = cart.id
    }

    void testSaveOrders() {
        assert orderService
        def session = [userId: userId]
        def user = User.get(userId)
        def orders = new Orders()
        //下面的 3 行是为了给 orders 的相关属性赋值
        orders.receiverName = user.userName
        orders.phone = user.phone
        orders.address = user.address

        orderService.saveOrder(session, orders)       //如果失败，会抛出异常
        def cart = Cart.get(cartId)
        assertEquals(cart.status, CartStatus.ORDERED)//检查 cart 是否更新


        //现在当前用户已经没有 status 为 NEW 的 cart 了，此时，
        //parepareCart 方法创建的 cart，是不包含商品的，则订单保存会失败

        orders = new Orders()
```

```
            orders.receiverName = user.userName
            orders.phone = user.phone
            orders.address = user.address
            shouldFail({
                //此时需要抛出异常
                orderService.saveOrder(session, orders)
            })

        }
    }
```

9.2.3 对 Controller 进行测试

从效果上来说，对 Controller 进行测试是最有实际价值的。因为 Controller 最能体现出程序与需求中每个功能点的对应关系，也最能体现出系统与用户的交互操作。然而，在 Grails 中，对 Controller 进行测试，并不是在模拟浏览器的请求，而是通过创建模拟对象（Mock），使 Controller 运行在某些特定的环境参数下（如特定的 URL 参数、表单参数、Session 等），然后验证 Controller 执行后对环境产生的输出（如跳转、对 Session 的修改、返回给 GSP 的 model 等）。

这里不对每一个 Controller 的每一个 action 都进行测试，只选取几个比较典型的 action 进行介绍。首先，对 UserController 的 loginCheck 进行测试，以测试登录的验证逻辑是否正确。

```
    class UserControllerTests extends GroovyTestCase {
        void setUp() {
            def user = new User(userName:'liang' ,
                    password: 'aabbcc'.encodeAsPassword(),
                    email:'xyz@abc.com' ,
                    phone: '13810000000' ,
                    address: "bejing china" )
            assertNotNull(user.save(flash:true))

        }
        void testLoginSuccess(){
            def ucPass = new UserController()
            ucPass.params.email = 'xyz@abc.com'     //模拟表单提交的 email
            ucPass.params.password = 'aabbcc'       //模拟表单提交的 password
            ucPass.loginCheck() //执行 action
            //检测 action 执行完成后要跳转到的 URL
            assert ucPass.response.redirectedUrl == '/goods/list'
            //检测 action 执行完成后要 session 的内容
            assert ucPass.session.userId
        }
```

```
    void testLoginFail(){
        def ucFail = new UserController()
        ucFail.params.email = 'xyz@abc.com'
        ucFail.params.password = 'incorrect'
        ucFail.loginCheck()
        assert ucFail.response.redirectedUrl == '/user/login'
        assertNull( ucFail.session.userId )
    }
}
```

UserControllerTests 包含两个测试点，分别对应登录成功和登录失败的两个场景。通过修改 Controller 实例的 params 属性的值，可以模拟通过表单向 Controller 提交数据。然后调用 action，并在 action 执行完毕后，校验此时要跳转到的目标 URL 和 session 的内容。

如果 action 中使用了 CommandObject，仍然可以使用 params 属性去指定参数。例如，对注册逻辑进行测试：

```
testUserSave(){
    def uc = new UserController()
    uc.params.password = ...
    uc.params.passwordAgain = ...
    …
    uc.save()
    …
}
```

如果 Controller 中使用了 Service，则在 test case 中需要为 Controller 的 Service 进行初始化。道理很简单，用 new 创建 Controller，不存在于 Spring 容器中，因而也不会被 Spring 所初始化。

要初始化 Controller 中的 Service，其实是很容易的，只需要在 test case 中定义对应的 Service 属性。从上一小节可以知道，Spring 容器会为 test case 注入 Service，因而将 Spring 容器中的 Service 实例手动注入 Controller 即可。例如，对 OrdersController 进行测试：

```
class OrdersControllerTests extends GroovyTestCase {
    def orderService    //由 Spring 容器注入 orderService
    void testSaveOrder(){
        def ordersController = new OrdersController()
        //在 test case 中为 Controller 初始化 service
        ordersController.orderService = orderService
        …
    }
}
```

下一个例子将讨论如何验证 action 向 GSP 传递的 Model 数据，这对于查询类的业务是很有实用价值的。

通过前面章节的学习可以知道，Controller 向 GSP 传递数据有两种方式：一种是使用 action 的返回值，此时 action 的返回值将传递给与之同名的 GSP 页面；另一种是使用 render 方法，通过 View 指定 GSP 页面，通过 Model 指定要传递的数据。如果是用第一种方式向页面传递数据，那么毫无疑问，通过 action 的返回值即可得到相应的数据；如果是使用第二种方式，通过访问 Controller 的 modelAndView 属性，即可得到 Model 和 View 的信息。

```
class GoodsControllerTests extends GroovyTestCase {
    void setUp() {
        def category = new Category(categoryName:'book')
        assertNotNull(category.save(flash:true))

        def book1 = new Goods(title:'book1', price:20.5 ,
            description:'', photoUrl:'', category: category )
        assertNotNull(book1.save(flash:true))

        def book2 = new Goods(title:'book2', price:10.5 ,
            description:'', photoUrl:'', category: category )
        assertNotNull(book2.save(flash:true))
    }
    void testListGoods() {
        //list action 需要用返回值获取 Model
        def goodsController = new GoodsController()
        def model = goodsController.list()
        assert model.goodsInstanceList.totalCount == 2
    }
    void testSearchGoods() {
        //search action 需要用 modelAndView 属性获取 Model
        def goodsController = new GoodsController()
        goodsController.params.categoryName = 'book'
        goodsController.params.title = 'book1'
        goodsController.search()
        def model = goodsController.modelAndView.Model
        assertEquals (model.goodsInstanceList.totalCount ,1)
    }
}
```

这里要补充说明一点，在测试时调用 Controller 的 action，并不会触发 Controller 的拦截器 inteceptor，也不会触发 filter。如果需要对它们进行测试，用户需要在 test case 中手动调用 Controller 拦截器或 filter 的闭包。

9.2.4　对 Taglib 进行测试

虽然读者还没有学习如何自己定义标签，但至少可以对 Grails 已经提供的标签进行测试。Grails 提供了一个名为 GroovyPagesTestCase 的类，通过它可以轻松地对标签库进行测

试。相应地，test case 也不再继承于 GroovyTestCase，而是继承 GroovyPagesTestCase。GroovyPagesTestCase 类提供了两个实用的方法：applyTemplate 和 assertOutputEquals。

applyTemplate 有两个参数：第一个参数为标签的内容，如“<g:link action="show">show</g:link>”；第二个参数为要传入的数据，是一个 map（回忆一下上一章介绍的 render 标签）。具体使用如下：

```
def template =
'<g:link controller="goods" action="show" id="${id}">show</g:link>'
def output = applyTemplate(template,[id:1])
assertEquals('<a href="/goods/show/1">show</a>',output)
```

assertOutputEquals 有 4 个参数：第一个参数为预期的输出；第二个参数和第三个参数分别与 applyTemplate 的第一个参数和第二个参数相同；第四个参数为可选，要求传入一个闭包，以实现对输出进行格式处理。具体使用如下：

```
class FormatTagLibTests extends GroovyPagesTestCase {
    void testDateFormat() {
        def template =
        '<g:dateFormat format="dd-MM-yyyy" date="${myDate}" />'
        def testDate = … // create the date
        assertOutputEquals('08-08-2008', template, [myDate:testDate],
            {it.toString().trim()})
    }
}
```

9.3 本章小结

本章从自动化测试的基础知识开始入手，结合实例，分别介绍了使用 Grails 对 Domain、Service、Controller、Taglib 等进行测试的方法。通过本章的学习，读者可以自行开发自动化的测试程序，进而实现测试驱动的开发。读者通过学习本章的内容，有利于养成良好的开发习惯，从而开发出质量更高的应用程序。

第10章 部署应用

使用 Grails 开发的网站，最终都要部署在生产环境中。人们当然不会希望在生产环境中使用 grails run-app 命令去执行程序，通常也不会选用 Grails 自带的 Jetty 作为 Web Server。因此，本章讨论如何将 Grails 的应用程序部署到生产环境中。

10.1 Grails 对部署的支持

Grails 的 run-app 命令实际上是可以使用运行环境参数的。例如，grails prod run-app 表示在生产环境运行。此时，程序的性能会有一定的提高，但仍然是使用 Jetty 作为 Web Server 运行的。如果部署在 Tomcat 这样的 Web Server 下，Web 应用的性能还会有很大的提升。因此，在真正的生产环境中，应选择更为专业的 Web Server 进行部署。使用 grails prod run-app 运行程序更多是帮助程序开发人员重现生产环境中发现的 bug：如果在开发模式下 bug 不能被重现，可以尝试在生产模式下运行。

Grails 对部署提供了很好的支持，可以使用 war 命令创建 Web 应用的 war 包。了解 Java Web 开发的人一定不会对 war 包陌生。通过标准的 war 包，可以很容易地将 Grails 的应用部署到 Tomcat、JBoss、GlassFish、WebSphere 等开源或商用的应用程序服务器中。

首先，运行 grails war 命令。

```
>grails war

Welcome to Grails 1.0.4 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: D:\grails-1.0.4

Base Directory: D:\workspace\GDepot
...
...
    [mkdir] Created dir: D:\workspace\GDepot\staging\WEB-INF\plugins
     [copy] Warning: D:\workspace\GDepot\plugins not found.
      [jar] Building jar: D:\workspace\GDepot\GDepot-0.1.war
   [delete] Deleting directory D:\workspace\GDepot\staging
Done creating WAR D:\workspace\GDepot/GDepot-0.1.war
```

命令的执行需要几秒钟的时间。命令执行完毕后，会生成一个名为 GDepot-0.1.war 的文件，这个文件就是包含了全部 GDepot 应用程序的 war 包。如果想修改 war 包的版本号，可以修改项目根目录下的 application.properties 文件。如果想指定默认的 war 包文件名，可以在 Config.groovy 文件中加入一行：

```
grails.war.destFile = "gdepot-prod.war"
```

要将 war 包部署到应用程序服务器下面，是很容易的事情。对于 Tomcat，有两种方式：一种是将 war 文件保存到 Tomcat 安装目录的 webapps 目录下，然后重启 Tomcat，会发现 Tomcat 将自动把 war 包解压，并且运行应用；另一种方式是使用 Tomcat 的管理控制台，在浏览器输入 http://localhost:8080/manager/html，然后在 Deploy 区域选择本机的 war 包，或者上传 war 包，然后单击 Deploy 按钮，这样就可以实现在远程计算机上部署 war 包了，如图 10-1 所示。

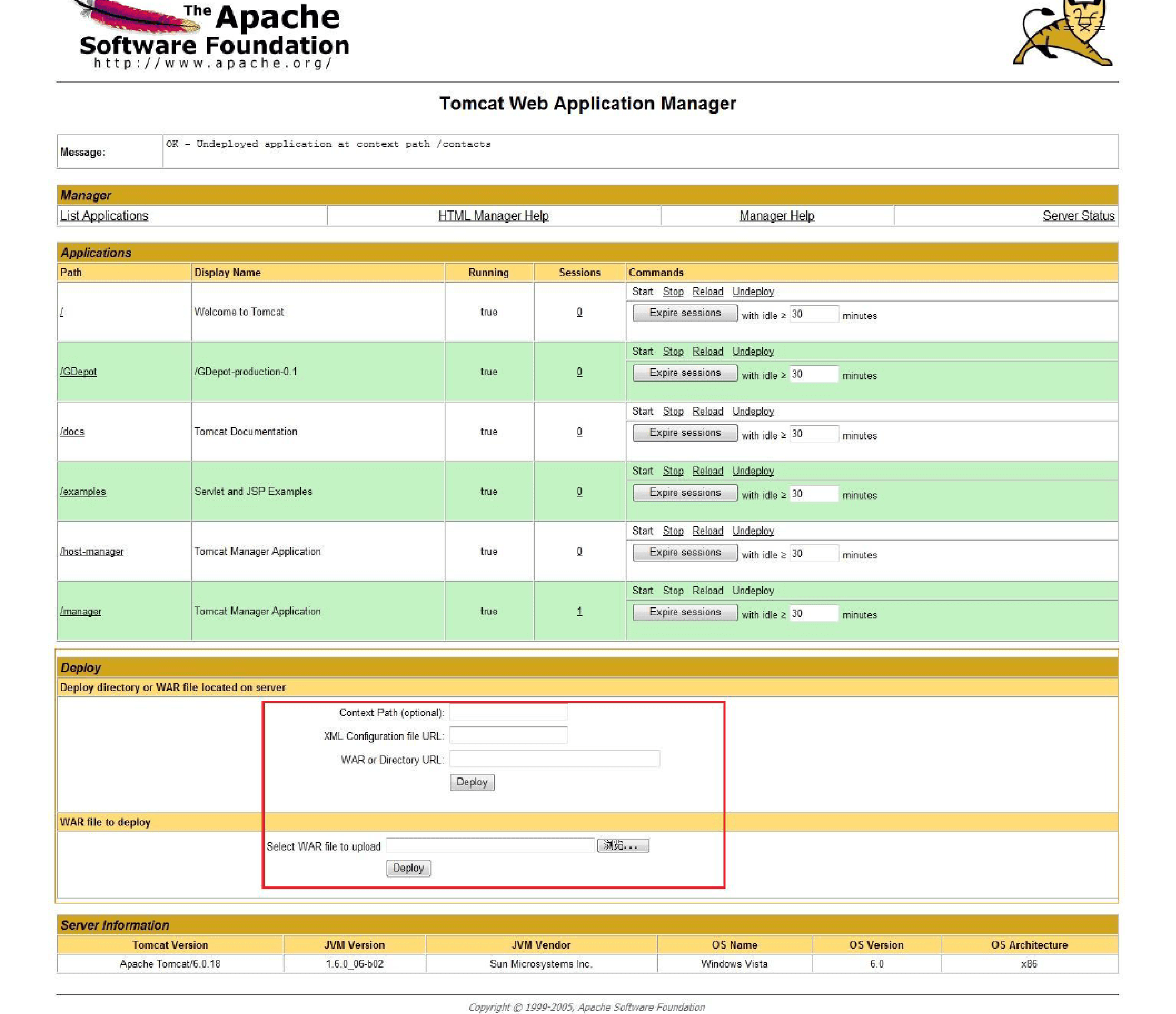


图 10-1　使用 Tomcat 管理控制台部署

对于其他更高级的应用服务器，部署方式也是类似的。引用 Grails 官方的数据，Grails 在不同应用服务器的部署测试结果如表 10-1 所示。

表 10-1 Grails 对 Web Container 的兼容情况

Container	Grails 1.0
Tomcat 5.5	Y
Tomcat 6.0	Y
Geronimo 2.0.2	Unknown, but see WAS CE below
Geronimo 2.1.1	Y
GlassFish v1（Sun AS 9.0）	Y
GlassFish v2（Sun AS 9.1）	Y
GlassFish v3	Y
Sun App Server 8.2	Y
Websphere 6.1	Y
Websphere Application Server Community Edition 2.0（WAS CE）	Y
Resin 3.2	Y
Oracle AS	Y
JBoss 4.2	Y
Jetty 6.1	Y
SpringSource Application Platform 1.0 beta	Y
Weblogic 8.1.2	N
Weblogic 10	Y

10.2 配置应用程序

虽然 Grails 努力追求的目标是零配置，但理想总是与现实存在一定差距。首先，数据库的配置就是不可避免的。在第 2 章已经介绍过，修改 DateSource.groovy 的内容，可以配置数据库的连接方式。但对于部署在生产环境下的应用而言，这里就可能存在一个问题。

通常情况下，负责开发项目的和部署项目的可能不是同一个人，而且很有可能不允许程序开发人员直接访问生产环境的数据库，即生产环境的数据库所使用的用户名和密码对开发人员是保密的。在这种情况下，就需要由部署人员去配置程序的数据库连接信息。但是，从 Grails 打包的 war 包中可以看到，DataSource.groovy 文件已经被编译成了一个 Java 的字节码文件（class 文件），如图 10-2 所示。因此，修改 DataSource.groovy 已经不是一件容易的事情了。

要解决这个问题，有以下几种思路。

1．使用应用服务器管理数据源

大部分的 J2EE 应用服务器都提供了完善的数据库连接池支持。应用程序可以通过 JNDI 使用由服务器提供的数据库连接池。在 Grails 中，访问 JNDI 数据源非常简单，只需要指定 dataSource 的 jndiName 属性即可，此时 DataSource.groovy 应写成如下样式：

DataSource.class
DataSource$_run_closure1.class
DataSource$_run_closure2.class
DataSource$_run_closure3.class
DataSource$_run_closure3_closure4.class
DataSource$_run_closure3_closure4_closure7.class
DataSource$_run_closure3_closure5.class
DataSource$_run_closure3_closure5_closure8.class
DataSource$_run_closure3_closure6.class
DataSource$_run_closure3_closure6_closure9.class

图 10-2　DataSource 被编译成了 Java 字节码

```
dataSource {
    //此时不存在全局配置信息
}
hibernate {
    cache.use_second_level_cache=true
    cache.use_query_cache=true
    cache.provider_class=
     'com.opensymphony.oscache.hibernate.OSCacheProvider'
}
// environment specific settings
environments {
    development {
        dataSource {
            driverClassName = "com.mysql.jdbc.Driver"
            username = "root"
            password = "mysql"
            pooled = true
            dbCreate = "update" // one of 'create', 'create-drop','update'
            url = "jdbc:mysql:  //localhost:3306/GDepot_dev"
        }
    }
    test {
        dataSource {
            driverClassName = "com.mysql.jdbc.Driver"
            username = "root"
            password = "mysql"
            pooled = true
            dbCreate = "update" // one of 'create', 'create-drop','update'
            url = "jdbc:mysql:  //localhost:3306/GDepot_test"
        }
    }
    production {
        dataSource {
            jndiName = "java:comp/env/GDepotProdDB"
            //只在生产环境下使用 JNDI
```

```
            }
        }
    }
```

此时 jndiName 的取值取决于应用服务器的配置，相应地对数据库的访问控制（用户名、密码）也只需要在应用服务器上进行配置，即 Grails 已不再管理数据库的连接。例如，在 Tomcat 下，需要通过在 server.xml 中加入如下配置代码：

```
<Context path="/GDepot" docBase="GDepot">
    <Resource name="GDepotProdDB" auth="Container"
        type="javax.sql.DataSource" maxActive="100" maxIdle="30"
        maxWait="10000" username="root"
        password="mysql" driverClassName="com.mysql.jdbc.Driver"
        url="jdbc:mysql://localhost:3306/gdepot_prod"/>
</Context>
```

不同应用服务器下配置数据源的方法并不相同，但基本原理相似，可以参考不同的服务器的文档进行配置。

2. 使用配置文件

Grails 中有一个读取内部或外部配置文件的统一入口。Config.groovy 中包含了大量的配置项，其中一项是默认注释掉了的 grails.config.locations 选项，通过该选项，可以实现从外部配置文件中读取配置信息。

```
// grails.config.locations = [ "classpath:${appName}-config.properties",
//             "classpath:${appName}-config.groovy",
//             "file:${userHome}/.grails/${appName}-config.properties",
//             "file:${userHome}/.grails/${appName}-config.groovy"]
```

它默认支持两种文件查找方式：一种是以当前的 classpath 为根目录，去查找配置文件；另一种是通过文件系统的绝对路径，去查找配置文件。grails.config.locations 是一个 list，因而可以一次指定多个外部配置文件。可以在外部配置文件中配置与 DataSource 相关的内容。Grails 会聪明地使用外部配置文件里的配置信息，覆盖 DataSource.groovy 中配置的信息。

首先将 Config.groovy 中的 grails.config.locations 定义为 grails.config.locations = ["classpath:datasource.properties"]。此时，Grails 会从 classpath 中寻找 datasource.properties 文件，然后在 grails-app\conf 文件夹下，创建 datasource.properties 文件[1]。在其中添加如下内容：

```
dataSource.driverClassName=com.mysql.jdbc.Driver
dataSource.username=root
dataSource.password=mysql
```

[1] Grails 在生成 war 包时，会自动将所有 Groovy 文件或 Java 文件编译成 Java 字节码（class 文件），其他与 Groovy 或 Java 在同一文件夹下的文件，则被复制到 WEB-INF\classes 中，即运行时的 classpath。

```
dataSource.pooled=true
environments.development.dataSource.dbCreate=update
environments.development.dataSource.url=jdbc:mysql://localhost:3306/Gde
pot_dev
environments.test.dataSource.dbCreate=update
environments.test.dataSource.url=jdbc:mysql://localhost:3306/GDepot_
test
environments.production.dataSource.dbCreate=update
environments.production.dataSource.url=jdbc:mysql://localhost:3306/Gdep
ot_prod
```

这里读者可以亲自体会一下 properties 文件中的每一个配置项与 DataSource.groovy 中的配置项的对应关系。

将配置信息写入到 properties 文件中，就不存在因为被编译而无法进行修改的问题了。当然，如果是将 properties 存放到 classpath 中，则每次更新 war 包都可能需要重新配置数据库。系统管理员可以事先与开发人员约定好，将数据库的配置文件存放到硬盘的某一固定目录下，然后开发人员在配置 grails.config.locations 时，使用绝对路径去指定配置文件。

需要补充说明的是，外部配置文件中不仅可以编写数据库的配置信息，还可以根据项目的需要，定义其他信息，如文件上传路径等。Grails 提供了简单易用的读取配置文件的方法，例如，在配置文件中定义了如下内容：

```
app.uploadpath=/var/www/Gdepot/upload
```

则可以在 Controller 中使用如下代码去读取配置文件的内容：

```
def path = grailsApplication.config.app.uploadpath
```

其中，grailsApplication 是 Controller 的一个属性（与 Service 对象一样，grailsApplication 是 Spring 注入到 Controller 中的）。

如果想在 Controller 以外的其他地方读取配置信息，则可以使用 ConfigurationHolder 类的静态属性 config，如：

```
def path = ConfigurationHolder.config.app.uploadpath
```

10.3　本章小结

本章从实用的角度，对部署 Grails 应用的方法进行了介绍。Grails 本身对部署提供了很好的支持。可以通过 grails war 命令，自动生成 war 包。使用标准的 war 包可以很容易地将应用程序部署在主流的 Web Server 上。此外，本章还介绍了在 Grails 中使用外部配置文件的方法。由系统管理员编写生产环境配置信息，在现实中比较常见。对于这种情况，使用外部配置文件显得更加实用。

第三篇

深入了解 Grails

通过前面两部分的学习，相信读者一定已经对 Grails 有了一定的了解，也能够使用 Grails 去开发 Web 应用了。但是，前面开发的例子多少都有一些理想化的成分，例如：使用 Grails 自动生成的数据库，不关心程序的性能。在这一部分将对 Grails 的一些细节进行更深入的讨论，以解决更多更实际和复杂的问题。

第11章 深入GORM

Grails 本质上是通过 Hibernate 进行数据库操作的，并且进一步地用 Groovy 语言对 Hibernate 进行了新的封装。封装后的框架还有一个很好的名字，叫 GORM。前面的章节已经对 GORM 进行了一定的介绍，如 list、get、delete、findBy*等方法，以及 HibernateCriteria Builder。本章将对 GORM 的一些高级特性进行较深入的讨论。本章的部分内容需要一定的 Hibernate 背景知识（文中也提供了一些参考文档的 URL，供读者查阅）。

11.1 自定义映射

进行数据库开发的一个很现实的问题就是，不可能完全由 Grails 去生成数据库的表结构。通常情况下，一些有经验且有发言权（通常是比较年长）的技术经理，更希望亲自进行数据库的设计。另一方面，如果是升级老系统，完全重建数据库也是不现实的。因此，如果 GORM 想要真正地被广泛使用，一定要能够被映射到已有数据库表上。这就是所谓的自定义映射。

11.1.1 基本映射

自 0.6 版开始，GORM 的自定义映射的功能得到了巨大的改进，并演化成为一种可以配置表、字段、缓存等的 DSL。具体进行映射其实非常简单：通过 mapping 闭包，就可以添加自定义的配置信息。

首先，对表名和字段名进行映射配置。

```
class Order {
    string receiverName
    Cart cart
    static mapping = {
        table 'orders'
        recevierName column:'Receiver_Name'
        cart column: 'cart_id'
    }
}
```

在 mapping 闭包中，使用“table '表名'”，可以配置 Domain 类对应的表名。使用 “属性名 column：'字段名'”，可以配置 Domain 类中某一属性，及其对应的数据库中的字段名。如果是与其他表关联的属性，则字段名为相应的外键字段。

对于字段而言，除了可以指定名称，还可以指定类型（这个更多用于让 GORM 自动创建表）。

```
class Goods {
    String description
    ...
    static mapping = {
        description type: 'text'
    }
}
```

“属性名 type：'类型名'” 可以为 Domain 类的属性指定数据库中对应的字段数据类型。其中可选的数据类型包括基本类型与自定义类型两大类，如表 11-1 所示。

表 11-1 基本类型与自定义类型

类型	列表
基本类型	integer, long, short, float, double, character, byte, boolean, yes_no, true_false string, text date, time, timestamp clob, blob, binary
自定义类型	实现了 org.hibernate.usertype.UserType 接口的类

更多关于自定义类型的内容，可以参考 Hibernate 的参考文档：http://www.hibernate.org/hib_docs/core/reference/en/html/mapping.html

11.1.2 配置主键

mapping 闭包中还可以定义数据库的主键。默认情况下，GORM 会为每个 Domain 添加一个默认 id，同时在数据库表中创建一个自增型的 id 字段作为主键。通过下面的代码，可以自己指定主键的字段名：

```
static mapping = {
    id column : 'student_id'
}
```

还可以通过 mapping 闭包指定主键的生成策略。下面的代码展示了使用 hilo 算法的主键生成策略（需要额外在数据库中创建 hi_value 表，该表必需包含 next_value 字段）：

```
static mapping = {
   id generator:'hilo',
    params:[table:'hi_value',column:'next_value',max_lo:100]
}
```

笔者推荐使用 uuid 方式生成主键：

```
class Goods {
    String id //uuid 方式生成的主键是字符串类型的
    ...
    static mapping = {
        id generator:'uuid'
    }
}
```

uuid 相比于其他的主键生成策略，有一个显著的优势，那就是主键的生成过程无需与数据库交互。这在出现高并发数据库写入时，会有较明显的性能优势。

除了 hilo 和 uuid 以外，Hibernate 还提供了多种主键生成策略，可以参考 Hibernate 的官方网站去了解相关内容[4]。

现在业内流行的数据库设计理念，不再推荐使用业务主键（业务相关的字段作为主键，如学号、身份证号、ISBN 等），而推荐使用逻辑主键（业务无关的字段作为主键）。原因是业务主键使用了业务数据，而业务相关的数据是无法保证不会发生变化的（例如，电话号码、学号、身份证号、ISBN 都是有可能升位、改格式等的），而一旦发生变化，则意味着大量级联更新，代价无疑是巨大的[1]。但是，对于老系统，多多少少可能已经使用了业务主键，并且相当一部分遗留系统中可能还使用了复合主键。为了实现对遗留系统的兼容，GORM 也对复合主键提供了支持：

```
class Student implements Serializable {
    String firstName
    String lastName
    String sid
    Date birthday

    static mapping = {
        id composite:['firstName', 'lastName']
    }
}
```

对于使用复合键的 Domain 类，应注意以下两点。

（1）使用复合键的 Domain 类，必需实现 Serializable 接口。

（2）使用主键从数据库中读取 Domain 实例（get 方法），可使用该类型对象的原型实例（主键包含的字段不为空的实例）作为 get 方法的参数，例如，从数据库取出名为张三的 Student：

```
def student = Student.get(new Student(firstName: '张', lastName: '三'))
```

[1] 可以想象，有海量数据的图书系统，如果 ISBN 作为主键，一旦 ISBN 要统一升位，几乎所有的数据库表都会受到影响，产生的级联更新操作将带来多么巨大的开销！

当然，如果不是为了兼容遗留系统的数据库，不推荐使用业务主键，更强烈不推荐使用复合主键。

11.1.3 “锁”与 Version

GORM 默认情况下将 Domain 配置为使用乐观锁（optimistic locking），乐观锁是 Hibernate 提供的一个特性：在数据库表中添加 version 字段并在 Domain 类中添加 version 属性，通过 version 记录数据被更新的次数。如果在执行更新时发现当前对象的 version 值与数据库中的 version 字段内容不一致，则会抛出异常（org.hibernate.StaleObjectStateException）。以下面的代码为例：

```
//首先取出对象
def goods = Goods.get(1)
//然后执行了某些操作，假设花费了较长的时间
goods.title = XXXX
//更新了对象的内容，然后保存
goods.save()
//如果保存时 goods 的 version 与数据库中的 version 不一致，
//说明在这段时间里，goods 被其他事务更新，当前操作就不应该再执行了。
//于是抛出异常
```

虽然乐观锁能提供不错的执行性能，但在一些情况下它并不适用，例如像上面的代码，需要用悲观锁（pessimistic locking）去解决问题。GORM 提供了 lock 方法去使用悲观锁：

```
def goods = Goods.get(1)
goods.lock() // lock for update
goods.title = XXXX
//其他修改 goods 的操作
goods.save()
```

多线程环境会给开发人员增加不少的负担。上面的代码只能保证程序在 lock 和 save 之间执行的时候，goods 的内容不会被其他线程修改。因此更严格的做法是直接用 lock 方法去读取数据，而不是用 get 方法：

```
def goods = Goods.lock(1)
goods.title = XXXX
//其他修改 goods 的操作
goods.save()
```

一旦事务提交，悲观锁的使命就完成了，Grails 会自动将其释放。由于本质上悲观锁是通过 SQL 语句 SELECT..FOR UPDATE 对数据库中的记录进行锁操作，对于不支持悲观锁的数据库（如 Grails 自带的嵌入式数据库 HSQLDB），lock 方法可能不能正确执行，所以使用前需要先确定所选择的数据库是支持悲观锁的。

在映射没有 version 字段遗留数据库时，可通过 mapping 闭包禁用 version 字段：

```
static mapping = {
    version false
}
```


11.1.4　事件与自动时间戳

有的时候会存在这样的需求：在删除 XXX 之前，要先对 XXX 进行 XXX 操作。GORM 给程序员提供了事件机制，可以分别在插入前、更新前、删除前、载入后触发并执行，如表 11-2 所示。

表 11-2　GORM 中可触发的事件

触发时机	实例

插入前触发	def beforeInsert = {
	//do something before insert
	}

更新前触发	def beforeUpdate = {
	//do something before update
	}

删除前触发	def beforeDelete = {
	//do something before delete
	}

载入后触发	def onLoad = {
	//do something after loaded
	}

如在 Domain 中定义了名为 lastUpdated 或 dateCreated 的属性，则 GORM 会自动在更新或插入时更新这两个属性的值，而无需程序员在 beforeInsert 和 beforeUpdate 这两个事件中去手动更新。但如果不希望 GORM 自动做这件事情，则应该在 mapping 闭包中禁用自动时间戳：

```
class Goods {
  Date dateCreated
  static mapping = {
    autoTimestamp false
  }
}
```

11.1.5　映射 Blob 字段

Blob 类型的字段在数据库中是比较特殊的字段，用于在数据库中存储二进制的文件。但 Blob 类型的字段使用通常较为特殊，不同的数据库对其支持也往往不全相同。因此，通常情况下使用数据库中 Blob 类型的字段并不容易。

然而，GORM 中对 Blob 类型的字段，提供了强大的支持，其使用过程也是难以置信的简单，只需在 Domain 中将该类型的字段定义为 Byte[]（此时的数据库表可能需要手动修改，本文以下代码将 photo 字段手动改成 MySQL 的 LongBlob 类型，以支持长度较大的文件）：

```
class Photo {
    byte[] photo
    String photoName
}
```

相应地，将上传文件写入数据库的 Controller action，可以为如下样式：

```
def upload = {
    def f = request.getFile("photo")
    def photo = new Photo()
    photo.name = f.originalFilename
    photo.photo = f.bytes
    photo.save()
}
```

很简单，不是吗？还是使用 save 方法，就可以将文件保存到数据库中，过程与普通的 Domain 没有任何区别。

相应地，将数据库中的文件取出，并且输出到浏览器显示的代码也很简单：

```
def showPhoto = {
    def p = Photo.get(params.id)
    response.contentType = "image/jpg"
    response.outputStream  << p.photo
}
```

由于不同数据库厂商提供的 Blob 字段不全相同，本文也无法保证上面的代码在所有的数据库上都能正常运行，仅在 MySQL 与 Microsoft SQL Server 2005（SQL Server 中的字段类型为 image）下测试并通过，其他的数据库笔者暂时没有条件测试，欢迎感兴趣的读者测试并共享测试结果。

11.1.6　定义非持久化属性

前面介绍的 Domain，其每个属性都是与数据库表中的字段相对应的。然而，有时人们会希望在 Domain 中定义一些不被数据库保存的字段。例如，用户注册时，需要用户输入两遍密码，但数据库中只需保存其中的一遍。回忆一下第 6.2 节，它是通过 CommandObject 实现的提交两次密码并验证是否相同。如果能够在 Domain 中声明某一属性无需保存到数据库中，则程序会得到明显的简化。

GORM 提供了 static 属性 transients，用于定义多个无需持久化的属性，如果使用 transients，则 User Domain 应该写为如下的形式：

```
class User {
    String userName
    String password
    String email
    String phone
    String address
    String passwordAgain
    static constraints = {
        userName(size:2..10,blank:false)
        password(size:6..30,blank:false)
        email(email:true,unique:true,blank:false)
        phone(matches:/\d{7,11}/,blank:false)
        address(maxSize:200,blank:false)
        passwordAgain(validator: {val, obj  ->
          if (val != obj.password ) return ['different.password']
        })
    }
    static transients = ['passwordAgain']
}
```

此时，无需 CommandObject 就可以实现用户注册中的“两次密码必需一致”的校验逻辑了。

11.2 深入理解 Domain 间的关系

11.2.1 一对一关系

一对一关系在关系数据库中属于最简单的情况。相应地，在 Grails 中是两个 Domain 类通过属性互相进行引用（可以是双向的，也可以是单向的）。

```
class Car {
    Engine engine
}
class Engine {
}
```

此时，Car 和 Engine 是一对一的关系，并且是单向的。通过 Car，可以很轻松地找到它对应的 Engine，但反过来则不容易通过 Engine 去找它对应的 Car。此时的数据库表结构中，表 Car 包含表 Engine 的 id 值作为外键，但表 Engine 中不包含表 Car 的 id。

如果同时还存在通过 Engine 去找 Car 这样的需求，则不妨建立双向的引用：

```
class Car {
    Engine engine
}
class Engine {
    Car car
}
```

此时生成的 Car 表和 Engine 表都包含对方的 id 作为外键，可以方便地实现通过 Car 取出 Engine 和通过 Engine 取出 Car。但此时的 Car 和 Engine 彼此不存在级联的更新与删除。如果希望实现级联更新与级联删除，需要按如下的方式去定义：

```
class Car {
    Engine engine
}
class Engine {
    static belongesTo = [ car: Car]
}
```

此时数据库的表结构不会发生变化，但却给 Engine 和 Car 定义了从属关系，Engine 属于 Car。如果 Car 被删除了，则 Engine 也会被级联删除（不禁想起一句话：皮之不存，毛将焉附）。同理，如果 Car 被更新，则相应的 Engine 也会被更新。

对于存在从属关系的两个对象，保存也更加容易。属主的一方（Car）可以保存另一方：

```
new Car(engine: new Engine()).save()     //此时只需调用 car.save 而无需调用
                                          engine.save
new Engine(car: new Car()).save()        //反过来则会报错!
```

11.2.2　一对多关系

一对多关系是关系数据库中最常见的关系。在数据库中，通过主键和外键进行映射；在 GORM 中，通过 hasMany 进行定义。前面例子中的 Cart 和 LineItem 就是典型的一对多关系：

```
class Cart {
    static hasMany = [lineItems : LineItem]
}
class LineItem {
    static belongsTo = [cart : Cart]
    ...
}
```

上面的例子中，Cart 与 LineItem 是一对多的关系，同时，它们之间的引用方式是双向的，并且可以通过 Cart 对 LineItem 进行级联更新和级联删除。

考虑到数据库中表的真实情况，如果 Cart 与 LineItem 是一对多的关系，则 Cart 表中实际上不包含 LineItem 表中的内容，而 LineItem 表中要包含 Cart 表的主键作为外键引用。但是，如果 LineItem 表中对应多个 Cart 的属性（当然，这个在逻辑上不通），则可能产生二义性的问题。

Grails 官方网站中给出了一个例子可以很好地说明问题：每个飞机场有多架出港飞机和入港飞机，每架飞机有一个起飞机场和一个降落机场，如图 11-1 所示。用 Domain 类描述飞机还是比较简单的：

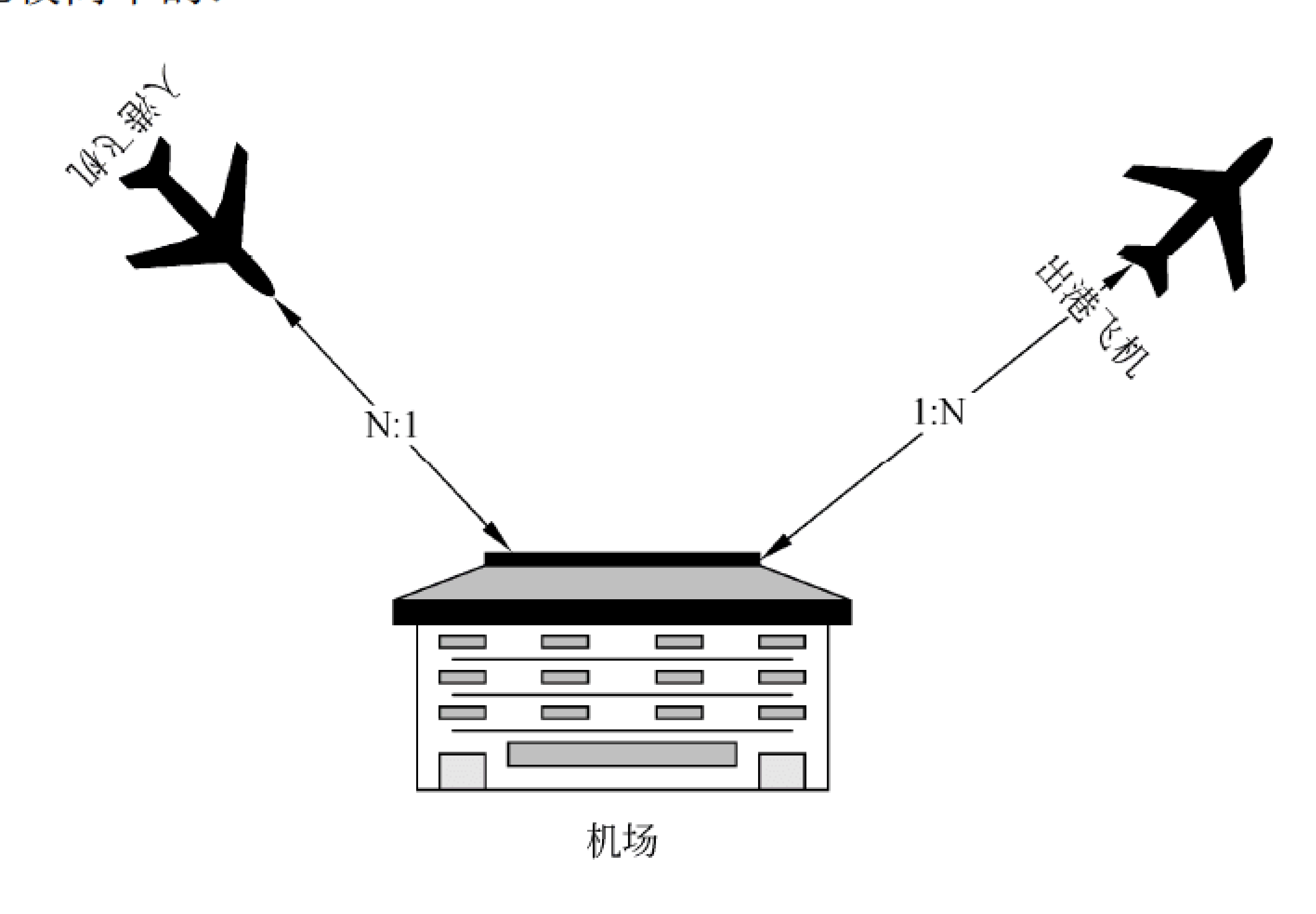


图 11-1　机场与飞机

```
class Flight {
    Airport departureAirport
    Airport destinationAirport
}
```

但是描述 Airport 有多个出港航班和多个入港航班，则有些困难。因为仅通过 hasMany 是无法说明 Airport 的多个航班中哪些是出港的、哪些是入港的。

这个问题比较复杂，但也比较普遍。Grails 提供了 mappedBy 去解决类似的问题。通过 mappedBy 可以明确属性名与字段的对应关系。例如，Airport 的定义方法如下：

```
class Airport {
    static hasMany = [outboundFlights:Flight, inboundFlights:Flight]
    static mappedBy = [outboundFlights:"departureAirport",
        inboundFlights:"destinationAirport"]
}
```

相应地，数据库中的表结构还是比较简单的：

```
CREATE TABLE 'airport' (
  'id' bigint(20) NOT NULL auto_increment,
  'version' bigint(20) NOT NULL,
  PRIMARY KEY  ('id')
```

```
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
CREATE TABLE 'flight' (
  'id' bigint(20) NOT NULL auto increment,
  'version' bigint(20) NOT NULL,
  'departure_airport_id' bigint(20) NOT NULL,
  'destination_airport_id' bigint(20) NOT NULL,
  PRIMARY KEY  ('id'),
  KEY 'FKB4318470C794BDE5' ('departure airport id'),
  KEY 'FKB43184708CE56E8B' ('destination airport id'),
  CONSTRAINT 'FKB43184708CE56E8B' FOREIGN KEY ('destination_airport_id')
  REFERENCES 'airport' ('id'),
  CONSTRAINT 'FKB4318470C794BDE5' FOREIGN KEY ('departure_airport_id')
  REFERENCES 'airport' ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

11.2.3 多对多关系

多对多关系在关系数据库中，属于实体间最复杂的情况，实现时需要使用一张关系表，通过两个一对多的关系，最终构成多对多关系，如图 11-2 所示。

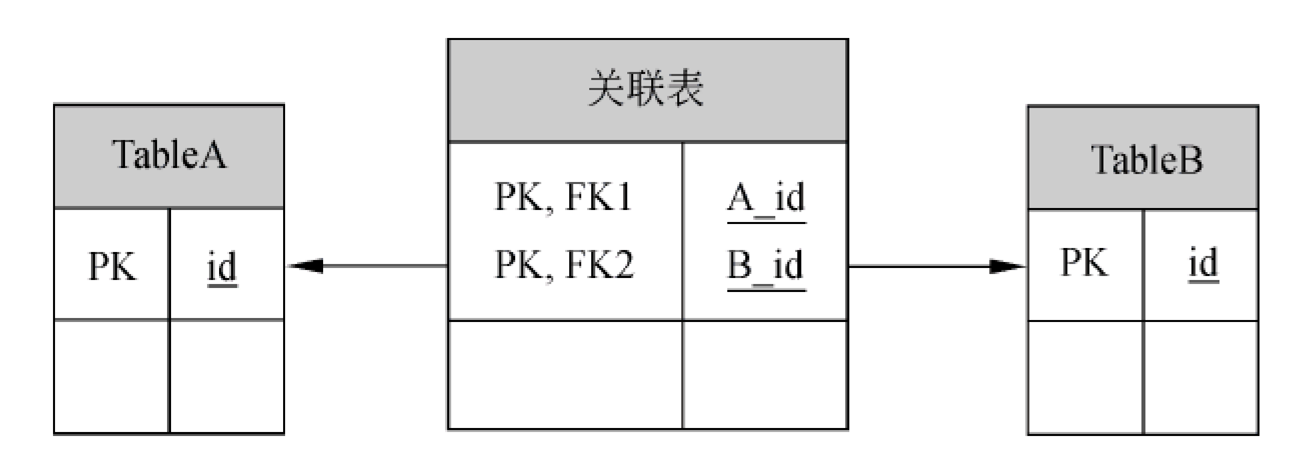


图 11-2 关系数据库中的多对多关系

在 GORM 中，同样可以使用类似的方式，分别创建对应上面 3 张表的 Domain 类，分别与 Domain 类建立双向的一对多关系。这种做法虽然简单易理解，但并不优雅。使用 GORM 强调的是 OO 建模，关联表对应的对象在逻辑上没有明确的含义，而且使用起来也不够方便。

GORM 为多对多关系提供了非常良好的支持，可以直接在两个对象上同时使用 hasMany。但是有一点不同的是，在多对多的情况下，必需在两者之间通过 belongsTo 定义从属关系。

```
class Student {
    static hasMany = [courses : Course]
}
class Course{
    static belongsTo = [Student] //仅用于指定从属关系
    static hasMany = [students: Student]
}
```

GORM 会自动创建相应的关联表。如果希望自己指定关联表，可以在 mapping 闭包中指定，例如：

```
class Student {
    static hasMany = [courses : Course]
    static mapping = {
        courses column:'Studnet_Id',joinTable:'Student_Course'
    }
}
class Course{
    static belongsTo = [Student] //仅用于指定从属关系
    static hasMany = [students: Student]
    static mapping = {
        students column:'Course_Id',joinTable:'Student_Course'
    }
}
```

对于多对多关系的 Domain 对象，其保存方式也比较特殊，需要用 addTo*方法（这也是个名称名可变的方法），例如：

```
new Student()
    .addToCourses(new Course())
    .addToCourses(new Course())
    .save()
```

因为 Student 是 Course 的属主，所以可以只调用 Student 的 save 而无需调用 Course 的 save。反过来则不成立，Student 不会被保存到数据库中。

```
new Course()
    .addToStudents(new Student())
    .addToStudents(new Student())
    .save()
```

当然，如果 Course 和 Student 都已经存在于数据库中（已经保存过或者是从数据库中读取出），上面两种写法都可以正确工作，例如：

```
new Student()
    .addToCourses(Course.get(xxx1))
    .addToCourses(Course.get(xxx2))
```

或

```
new Course()
    .addToStudents(Student.get(xxx3))
    .addToStudents(Student.get(xxx4))
```

与 addTo*方法使用相似但是功能相反的有 removeFrom*方法，用于解除对象间的关系

（删除关联表中的数据）：

```
def student = Student.get(1)
def course = Course.get(1)
student.removeFromCourses(course)
```

11.2.4 继承关系

继承是 OO 技术中常见的一种形式，但在关系数据库中，并不存在继承的概念。幸运的是，GORM 提供了一种机制，可以弥合这两种不同的技术体系的差别[1]。

对于继承，GORM 支持两种继承策略：默认为多个子类共用同一张表，此时，数据库相关的类共同使用一张“最大的”表，它包含每一个子类的每一个属性；另一种策略是所有的子类分别使用单独的一张表，每张表和每个子类是一一对应的关系。下面的代码是第一种继承策略的示例：

```
class People {
    String name
    int age
}
class Student extends People {
    String studentId
}
```

默认情况下，该代码生成数据库表的 DDL：

```
CREATE TABLE 'people' (
  'id' bigint(20) NOT NULL auto_increment,
  'version' bigint(20) NOT NULL,
  'age' int(11) NOT NULL,
  'name' varchar(255) NOT NULL,
  'class' varchar(255) NOT NULL,
  'student_id' varchar(255) default NULL,
  PRIMARY KEY  ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

其中的 class 字段用于识别每条记录所对应的 Domain 类型。执行下面的代码保存两条记录到数据库中：

```
new People(name: "John",age:20).save()
new Student(name: "Tom",age:20,studentId: "S201").save()
```

则数据库中数据记录变为如表 11-3 所示。

[1] 准确地说，是 Hibernate 提供的机制。

表 11-3　people 表中的数据

id	version	age	name	class	student_id
1	0	20	John	**People**	null
2	0	20	Tom	**Student**	S201

这种“基类和子类共用一张数据库表”的实现策略叫做 table-per-hierarchy，它要求子类的属性在数据库中的字段可以为空(nullable)。

另一种“基类和子类分别使用不同的数据库表”的实现策略叫做 table-per-subclass。需要在基类使用 mapping 闭包进行定义：

```
class People {
    String name
    int age
    static mapping = {
        tablePerHierarchy false
    }
}
```

此时，数据库表的 DDL 为：

```
CREATE TABLE 'people' (
  'id' bigint(20) NOT NULL auto_increment,
  'version' bigint(20) NOT NULL,
  'age' int(11) NOT NULL,
  'name' varchar(255) NOT NULL,
  PRIMARY KEY  ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE 'student' (
  'id' bigint(20) NOT NULL,
  'student id' varchar(255) default NULL,
  PRIMARY KEY  ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

可以看到，GORM 创建了两张数据库表。子类 Student 所对应的表的主键不是增类型的。再次执行前面的数据保存，则表中数据如表 11-4、表 11-5 所示。

表 11-4　people 表中的数据

id	version	age	name
1	0	20	John
2	0	20	Tom

表 11-5　student 表中的数据

id	student_id
2	S201

显然，Table-per-hierarchy 的方式可以消除冗余字段，数据库存储效率更高一些。然而，每次对子类对象的查询，都需要对两张数据库表进行联合查询，性能会有一定的下降。因

此，在使用 table-per-subclass 时，需要先分析业务的性能需求并做到避免设计继承深度过深的 Domain 类。

使用 OO 理念设计的 Domain 类，可以进行多态查询，使用起来十分方便，例如：

```
People.list()//返回全部，包括 People，Student，因为 Student 也是 People
Student.list()//只返回 Student
```

11.2.5 合成关系

利用合成关系，可以将一个类“嵌入”到另一个类中去，从而减少不必要的表联接，例如：

```
class Rectangle{
    Point startPoint
    Point endPoint
    static embedded = ['startPoint', 'endPoint']
}

class Point {
    int x
    int y
}
```

生成的 Rectangle 表的 DDL 为：

```
CREATE TABLE 'Rectangle' (
  'id' bigint(20) NOT NULL auto_increment,
  'version' bigint(20) NOT NULL,
  'start_point_x' int(11) NOT NULL,
  'start_point_y' int(11) NOT NULL,
  'end_point_x' int(11) NOT NULL,
  'end_point_y' int(11) NOT NULL,
  PRIMARY KEY  ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

如果不希望 Point 类也被单独地生成一张数据库表，则不要将 Point 类的内容写到一个单独的 Groovy 文件中，而应该直接将其写在 Rectangle 类所在的 Groovy 文件中。

11.3 数据库查询小结

11.3.1 GORM 提供了便捷的查询方法

GORM 提供了许多简单易用且功能强大的方法用于对数据库进行查询操作，前面的章

节里已经进行了介绍，这里仅做简单的小结。GORM 提供的查询方法可以划分为 8 个系列，同一系列的方法具有相似的逻辑，如表 11-6 所示。

表 11-6　GORM 查询方法总结

方法系列	说明

list listOrderBy*	对整表进行查询，无法指定查询条件，但是可以使用分页 Map 进行分页和排序。
	例：
	Goods.list()//返回全部 Goods 的列表
	Goods.list([max:10,offset:10,sort: 'title', order: 'asc']) //使用分页 Map
	//进行分页和排序
	
	Goods.listOrderByTitle([max:10,offset:10,order: 'desc'])

get
getAll
	根据 Domain 的 id（数据库主键，若为复合主键，使用其自身实例）查询。
	例：
	get 方法是使用一个 id，返回一条记录，如:
	def cart = cart.get(1) //返回 id 为 1 的 cart
	
	getAll 方法是使用多个 id（id 的 List），返回多条记录，
	如：
	def cartList = Cart.getAll([1,2,3,4]) //返回 id 为 1、2、3、4 的 4 条 Cart 记录

findBy*
findAllBy*
	功能强大的查询方法，以属性值作为约束条件，也可以使用分页 Map 进行分页和排序，详见第 6 章。
	例：
	def goods = Goods.findByTitle('Grails') //返回 title 为 Grails 的一条记录
	def goodsList = Goods.findAllByTitle('Grails')// 返回 title 为 Grails 的
	//所有记录

count
countBy*
	查询数据库表中的记录条数，相当于使用 SQL: select count(*) from…
	例：
	Goods.count()//返回 Goods 的记录总数，相当于 select count(*) from goods
	Goods.countByTitle('Grails')//返回 title 为 Grails 的记录总数，相当于
	//select count(*) from goods where title = 'Grails'

findWhere
findAllWhere
	通过 Map 对数据库进行查询，多个条件之间只能是“与”的关系，而且不支持分页和排序。
	例：
	//返回一条 title 为 Grails 并且 price 为 20.0 的 Goods 记录
	Goods.findWhere([title: 'Grails', price: 20.0])
	
	//返回所有 title 为 Grails 并且 price 为 20.0 的 Goods 记录
	def goodsList = Goods.findAllWhere([title: 'Grails', price: 20.0])

createCriteria
withCriteria
	使用 HibernateCriteriaBuilder 对数据库进行查询，详见第 5 章。
	例：
	def c = Goods.createCriteria()
	def goodsList = c.list({ ...}) //{…}的内容为查询闭包
	
	def goodsList = Goods.withCriteria({ … }) //{…}的内容为查询闭包

续表

方法系列	说明

find
findAll
	使用 HQL 进行数据库查询，但 HQL 的查询内容仅限于当前 Domain 类。findAll 方法可以使用分页 Map 进行分页。
	例：
	//返回 HQL 执行结果的一条记录。
	def goods = Goods.find('from Goods where …')
	
	//返回 HQL 的全部执行结果。
	def goodsList = Goods.findAll('from Goods where …')

executeQuery executeUpdate	使用 HQL 操作数据库，对 HQL 的查询内容不加限制。executeQuery 用于执行查询，executeUpdate 用于执行更新。 如果使用了 Hibernate 的二级缓存，则应避免使用 HQL 更新数据库

说明：

（1）带*的方法属于动态方法，方法名可以变化。

（2）对于返回多条记录的查询，通常会返回 List 实例。

（3）分页 Map 中可包含 max、offset、sort、order 共 4 个键。其中 max 和 offset 用于控制分页，sort 和 order 用于控制排序。max 用于设置最大返回查询的结果，offset 设置查询的起始记录，sort 用于指定排序的属性（字段），order 用于指定排序的方式（升序或降序）。

11.3.2　基于 HQL 的查询

关于 GORM 的查询，前面的章节里已经介绍了很多，这里无需重复介绍，除了 list 方法、get 方法、findBy 方法、HibernateCriteriaBuilder 外，还有一种源于 Hibernate 的重量级的查询方法，那就是 HQL。

HQL 是面向对象的查询语言，语法与 SQL 相似，功能非常强大。但是相对于前面介绍过的查询方法，使用 HQL 是相对比较复杂的，一般应用于解决包含统计分析等复杂问题的场合。

下面举例说明，对于最简单的情况，查询取出所有的商品列表，可以用如下 HQL 进行查询：

```
from Goods
```

如果希望查询 category 为“book”的商品，则 HQL 为：

```
from Goods g where g.category.categoryName = 'book'
```

对于前面章节中计算购物车内商品总价的程序，完全可以用 HQL 在数据库内部进行计算：

```
select sum(l.itemNumber * l.goods.price) from LineItem l where l.cart.id=?
```

上面的 HQL 用到了查询参数，HQL 支持有名和匿名两种参数形式。如果是匿名查询，

应使用一个数组传递参数，数组内元素要与“?”按顺序一一对应；如果使用有名参数，则应使用 Map 去传递参数，Map 的 key 要与参数名一一对应（不用考虑顺序）。上例如果使用有名参数查询，HQL 的写法如下：

```
select sum(l.itemNumber * l.goods.price) from LineItem l where l.cart.id=:
cartId
```

HQL 所包含的内容异常丰富，但它并不是本书的重点，这里不对其进行进一步的介绍，感兴趣的读者可以参考 Hibernate 官方网站中 HQL 的教程[5]。

在 Grails 中执行 HQL 有两类方法：一类是 find、findAll 方法；另一类是 executeQuery、executeUpdate 方法。对于 find 和 findAll 方法，要求 HQL 只能对当前的 Domain 类所对应的表进行查询；而 executeQuery、executeUpdate 方法则不对查询目标做任何要求。因此，Cart 的 totalPrice 方法可以改写为如下形式（匿名参数或有名参数两种写法）：

```
def totalPrice() {
    if(this.id) {
        //匿名参数
        def result = Cart.executeQuery("\
            select sum(l.itemNumber * l.goods.price) \
            from LineItem l where l.cart.id=?" ,
            [this.id])

        //有名参数
        def result = Cart.executeQuery("\
            select sum(l.itemNumber * l.goods.price) \
            from LineItem l where l.cart.id=:cartId " ,
            [cartId:this.id])

        if(result)
            return result[0]
    }
}
```

注意，HQL 不支持 Groovy 的多行字符串“"""”，但可以使用“\”进行换行。

使用 HQL 同样支持分页 Map（只分页不排序），方法和 findBy 方法完全一致，只需要再传入一个描述分页逻辑的 Map，Map 中包含 max 和 offset（参考 findBy*的用法）：

```
Cart.executeQuery(hql, [params],[max:10,offset:0])
```

11.4 对 GORM 进行性能优化

前面的章节曾经介绍过，GORM 默认使用延时加载。但使用延时加载就可能带来 n+1 次查询的问题，从而带来严重的性能问题。针对 n+1 次查询的问题，有两种解决方案：一

类是取消延时加载或设置抓取模式为“eager”，从而实现一次查询取出全部对象；另一类是使用二级缓存，从而减少对数据库的访问。

11.4.1 设置抓取模式

对于存在关联的对象，可以根据具体使用情况，设置为非延时加载。对于前面讨论过的 LineItem 和 Goods，就应该使用非延时加载。显然，访问 LineItem 的时候，都要访问 Goods。使用 CriteriaBuilder 进行查询，可以指定抓取策略。

```
def c = LineItem.createCriteria()
def list = c.list {

    fetchMode("goods", org.hibernate.FetchMode.EAGER)
}
list.each {
    println it.goods.title
}
```

此时生成的 SQL 是对两张表进行连接后的查询：

```
    select
        this_.id as id1_1_,
        this_.version as version1_1_,
        this_.cart_id as cart3_1_1_,
        this_.goods_id as goods4_1_1_,
        this .item number as item5 1 1 ,
        goods2_.id as id4_0_,
        goods2 .version as version4 0 ,
        goods2_.category_id as category3_4_0_,
        goods2_.description as descript4_4_0_,
        goods2_.photo_url as photo5_4_0_,
        goods2_.price as price4_0_,
        goods2_.title as title4_0_
    from
        line item this
    inner join
        goods goods2
            on this_.goods_id=goods2_.id
```

11.4.2 使用二级缓存

Grails 自 1.0 版开始，对 Hibernate 的二级缓存提供了支持。Hibernate 的二级缓存分别是针对实体对象和 SQL 进行的（Query-Cache）。二级缓存相当于 Map，分为对象 Map 和

Query Map 两类。对象 Map 是以 Domain 类的 id 作为 key，将对象内容进行缓存；Query Map 是以 SQL 语句作为 key，但只缓存查询结果的 id。

首先要在 DataSource.groovy 中开启二级缓存：

```
hibernate {
    cache.use_second_level_cache=true
    cache.use_query_cache=true
    cache.provider_class=
        'com.opensymphony.oscache.hibernate.OSCacheProvider'
}
```

从 1.03 版开始，Grails 默认使用 OSCache 作为默认的缓存组件。感兴趣的读者可以访问 http://www.opensymphony.com/oscache/了解更多有关 OSCache 的信息。如果想要对 OSCache 进行详细的配置，可以编写 OSCache 的配置文件（oscache.properties），并把它放到 grails-app\conf 中。通过这个配置文件，可以设置 OSCache 组件的内存使用、硬盘使用、调度算法等。

可选用的缓存组件如表 11-7 所示。

表 11-7　Cache 组件（来自 Hibernate 官方网站）[1]

Cache	Provider class	类型	是否支持集群	是否支持 Query Cache
EHCache	org.hibernate.cache .EhCacheProvider	memory, disk	否	是
OSCache	org.hibernate.cache .OSCacheProvider	memory, disk	否	是
SwarmCache	org.hibernate.cache .SwarmCacheProvider	clustered （ip 广播）	是	否
JBoss TreeCache	org.hibernate.cache .TreeCacheProvider	clustered （ip 广播），transactional	是	是 （需要时钟同步）

如果要将应用部署到集群环境中，则需要选用一个支持集群的 Cache 组件。

首先，配置对象缓存，这样就可以为不同的 Domain 对象配置具体的缓存使用策略了。对 Domain 对象使用缓存有一个重要的前提条件，就是它的读取访问次数要远大于更新次数，否则使用缓存是没有意义的。对 Goods 类使用缓存，如下所示：

```
class Goods {
    static mapping = {
        cache true
    }
}
```

[1] 事实上，最新版的 EHCache 和 OSCache 都已经支持使用集群环境了，读者可以访问相关网站了解。

此时，使用的策略是“read-write”，并且对所有属性进行缓存。如果想进行详细配置，则可以写成：

```
class Goods {
    static mapping = {
        cache usage: 'read-write', include: ' all '
    }
}
```

usage 决定了使用缓存的策略，其中可选用的 Cache Usage 如表 11-8 所示。

表 11-8　Cache Usage

策略	说明
read-only	针对只读型的数据，要求 Domain 对象不能被更新（insert 和 delete 是可以的）。拥有最佳的缓存性能
read-write	允许被缓存的对象进行更新操作。实际测试中表现效果一般
nonstrict-read-write	允许被缓存的对象进行更新操作。性能较“read-write”有显著的提高，推荐使用。但由于使用了非严格读写的策略，不适合使用在存在多个事务并发访问的敏感的数据上（如金融、财务等）
transactional	要求缓存组件中的数据也拥有事务隔离能力（如 JBoss TreeCache），并且程序必须运行在完整的 JTA 环境中。如果使用了这一策略，还需要对 Hibernate 进行额外设置：需要在 DataSource.groovy 中的 Hibernate 闭包中指定 transaction.manager_lookup_ class。参考 Hibernate 官方网站中关于 Cache 配置的部分：http://www.hibernate.org/hib_docs/reference/en/html/performance.html#performance-cache

默认情况下，Cache 将 Domain 对象的所有属性都记入 Cache（即 include: 'all'）；也可以仅缓存 Domain 中非延时加载的属性（include: 'non-lazy'）。

除了在 Domain 上配置缓存，还可以为它的集合属性配置缓存：

```
class Cart {
    static hasMany = [lineItems:LineItem]
    static mapping={
        lineItems cache:[ usage:'nonstrict-read-write']
    }
}
```

在 Congfig.groovy 中开启 Hibernate 对 Cache 的日志输出，以观察 Cache 的使用情况（建议使用 grails console，并重复执行 Domain.get()方法）：

```
logger {
        grails="error"
        StackTrace="error,stacktraceLog"
        org {
            codehaus.groovy.grails.web.servlet="error"  //  controllers
            codehaus.groovy.grails.web.pages="error" //  GSP
            codehaus.groovy.grails.web.sitemesh="error" //  layouts
            codehaus.groovy.grails."web.mapping.filter"="error"
            codehaus.groovy.grails."web.mapping"="error" // URL mapping
```

```
                codehaus.groovy.grails.commons="info" // core / classloading
                codehaus.groovy.grails.plugins="error" // plugins
                codehaus.groovy.grails.orm.hibernate="error"
                springframework="off"
                hibernate="off"
                hibernate {
                      cache = "debug"
                 }
            }
    }
```

观察日志的输出内容，可以看到使用了不同缓存策略的对象在查询时的缓存命中情况。

从输出的日志中会发现一个现象，Query-Cache 从来都没生效过。并不是配置有问题，而是因为在 Hibernate 中要使用 Query-Cache，需要设置 Query 或 Criteria 的 cacheable 属性为 true。所以，执行下面的查询代码，就可以在日志中看到 Query-Cache 也被命中了：

```
def c = Goods.createCriteria()
def list = c.list {
    cacheable(true)
}
```

查看日志中 Cache 相关的信息：

```
[24641] cache.StandardQueryCache checking cached query results in region:
org.hibernate.cache.StandardQueryCache
[24643] cache.StandardQueryCache Checking query spaces for up-to-dateness:
[goods]
[24644] cache.StandardQueryCache returning cached query results
[24645] cache.NonstrictReadWriteCache Cache lookup: Goods#1
[24645] cache.NonstrictReadWriteCache Cache hit
[24647] cache.NonstrictReadWriteCache Cache lookup: Goods#2
[24647] cache.NonstrictReadWriteCache Cache hit
```

由于 Query-Cache 只是缓存查询结果的 id（主键），当 Query-Cache 命中时，从缓存中取出的也只是这一组 id，如果 id 对应的对象不在二级缓存中，还需要从数据库中读取。因此，千万不要在查询没有配置缓存的 Domain 时使用 Query-Cache，这样将带来严重的性能问题。

目前唯一能够使用 Query-Cache 的办法，就是使用 HibernateCriteriaBuilder 进行查询，并且指定 cacheable 为 true。这是因为使用 GORM 提供的 list、getAll、find*等方法，Query-Cache 都不会起作用。Grails 的开发组织在封装上述方法时，分别使用了 Spring 的 Hibernate-Template 和 Hibernate 的 Criteria。但无论使用了哪种方式，都没有设置相关对象的 cacheable 属性为 true，因而根本没有使用 Query-Cache。

这不能不说是现在版本 Grails（Grails 1.0～Grails 1.0.4）的一个 bug，而且用户可以在

官方提供的用于跟踪 bug 的数据库中找到这个 bug[6]。本书的最后部分将尝试修改 Grails 的代码去解决这个 bug，这就是源代码开放的好处。[1]

缓存的使用并不是越多越好，需要具体问题具体分析。通过压力测试和日志分析，对系统进行反复的调整，才能最终将系统的性能调至最优。

11.5 使用 GRAG 工具生成 Domain

GRAG (GRails Application Generator)并不是 Grails 框架的一部分，它是一个用于生成 Grails Domain 类的工具。它可以读取数据库中的表结构，并根据表结构生成 Domain 类。有了 GRAG 工具的帮助，兼容旧数据库不再是一件麻烦的事情了。熟悉 Hibernate Tools 的读者应该对这种工具比较熟悉了。

首先下载 GRAG：http://sourceforge.net/project/showfiles.php?group_id=230641。下载完成后解压即可使用。Windows 平台下（假设解压到 d:\grag1.0）运行 d:\grag1.0\bin\gui.bat，Linux/UNIX 平台下（假设解压到~/grag1.0）运行~/grag1.0/bin/gui.sh，会看到如图 11-3 所示的窗口。

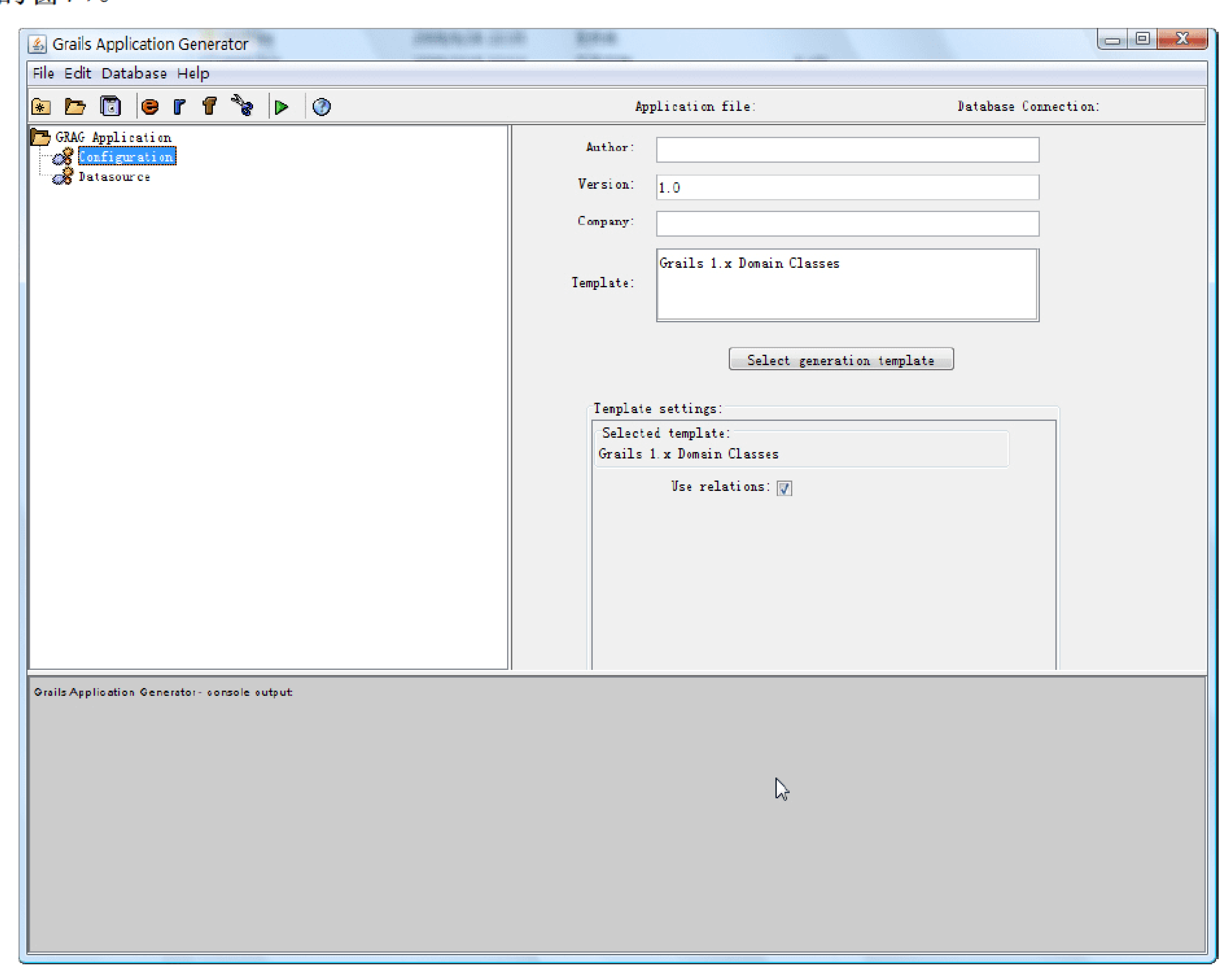


图 11-3 GRAG 启动界面

接下来保存工程（注意，这里的工程和 Grails 工程没有直接关系），执行 File|Save 命

[1] 事实上，现在最新的 Beta 版——1.1 Beta2，已经为分页 Map 中添加了 cache key，用于允许用户在查询时开启 Query-Cache。

令，弹出如图 11-4 所示的对话框，单击“保存”按钮。

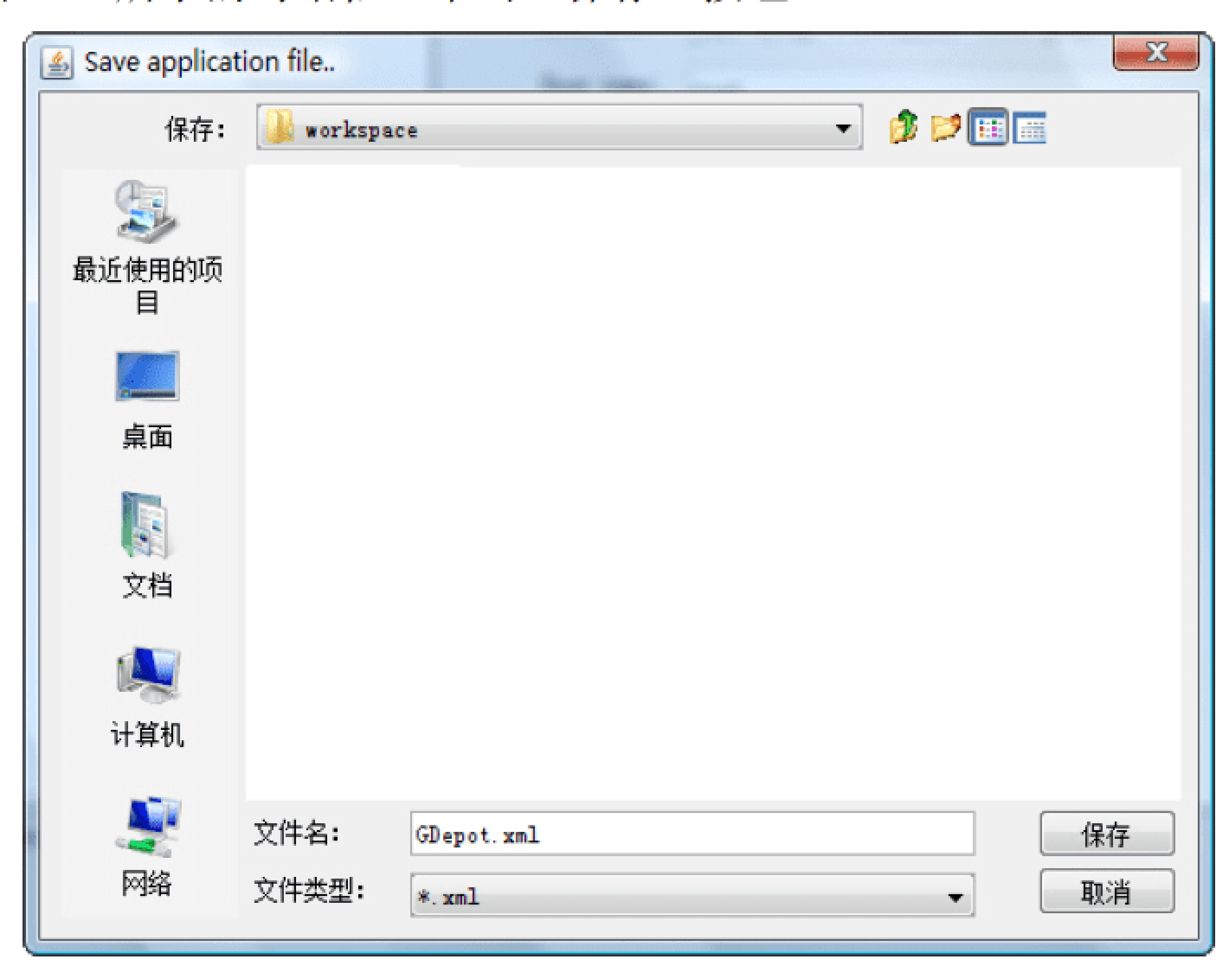


图 11-4　保存 GRAG 工程文件

接下来配置数据库的连接，单击左侧工程中的 Datasource，然后输入数据库的类型、连接 url、用户名和密码[1]，如图 11-5 所示。

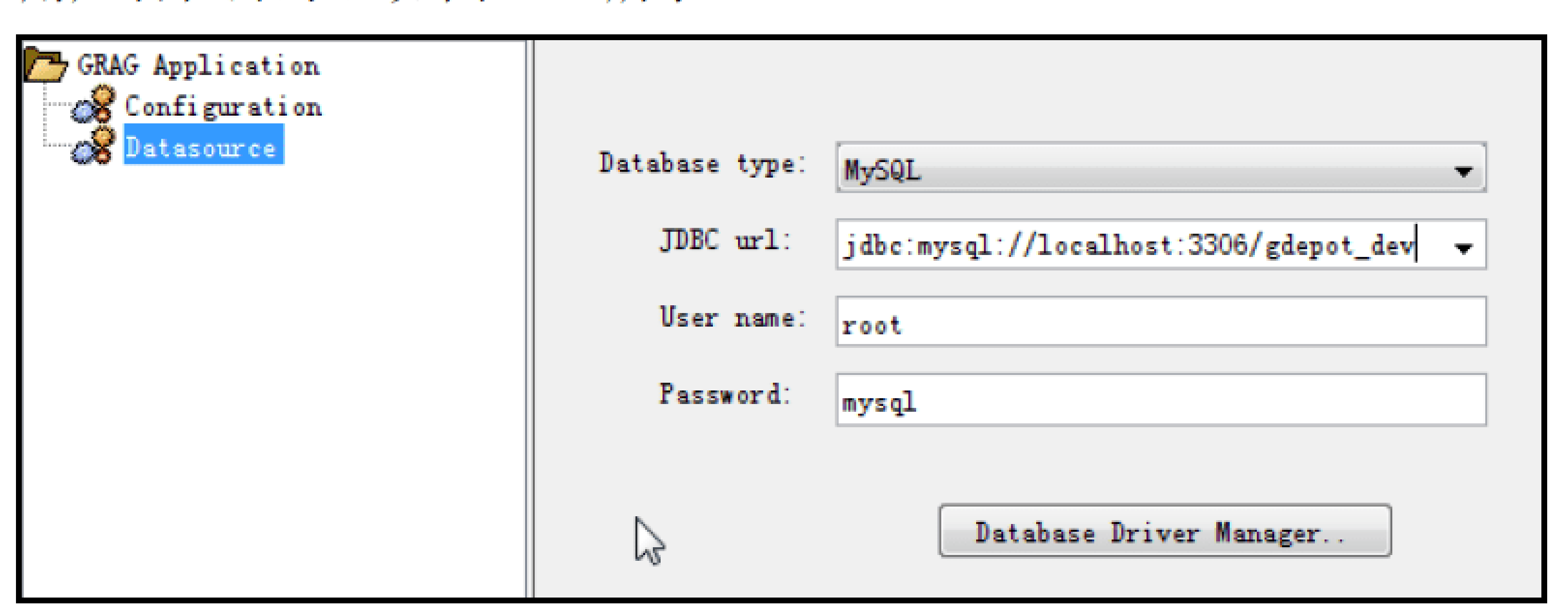


图 11-5　配置数据库连接

然后执行 Database|Create connection 命令，在弹出的对话框中修改连接参数，单击 Connect 按钮，如图 11-6 所示。

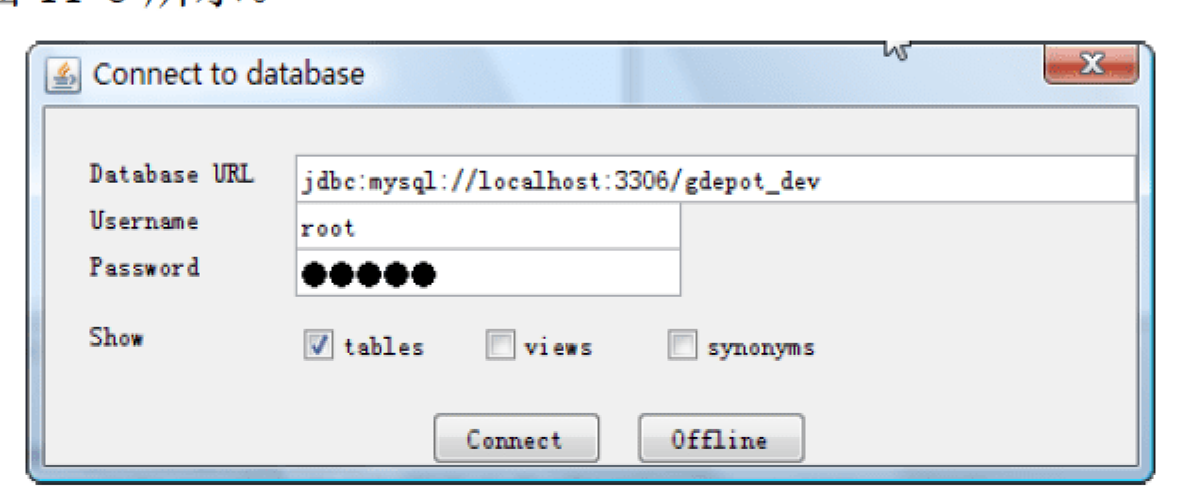


图 11-6　连接数据库

[1] GRAG 默认提供了 PostgreSQL、MySQL、Hypersonic SQL 共 3 种数据库的 JDBC 驱动，如果使用了其他数据库，还需要先单击 Database Driver Manager 按钮配置其他数据库的 JDBC 驱动。

连接成功后，执行 Edit|Add|entity 命令（或按 Ctrl+1 键），或者单击工具栏的按钮。

在弹出的对话框中选择需要映射的数据库表（不要选择多对多的关系表，因为它是不需要映射的），然后单击 Select 按钮，如图 11-7 所示。可以发现 GRAG 会将被选中的表配置为 Entity Class，并显示在左侧的工程中，如图 11-8 所示。

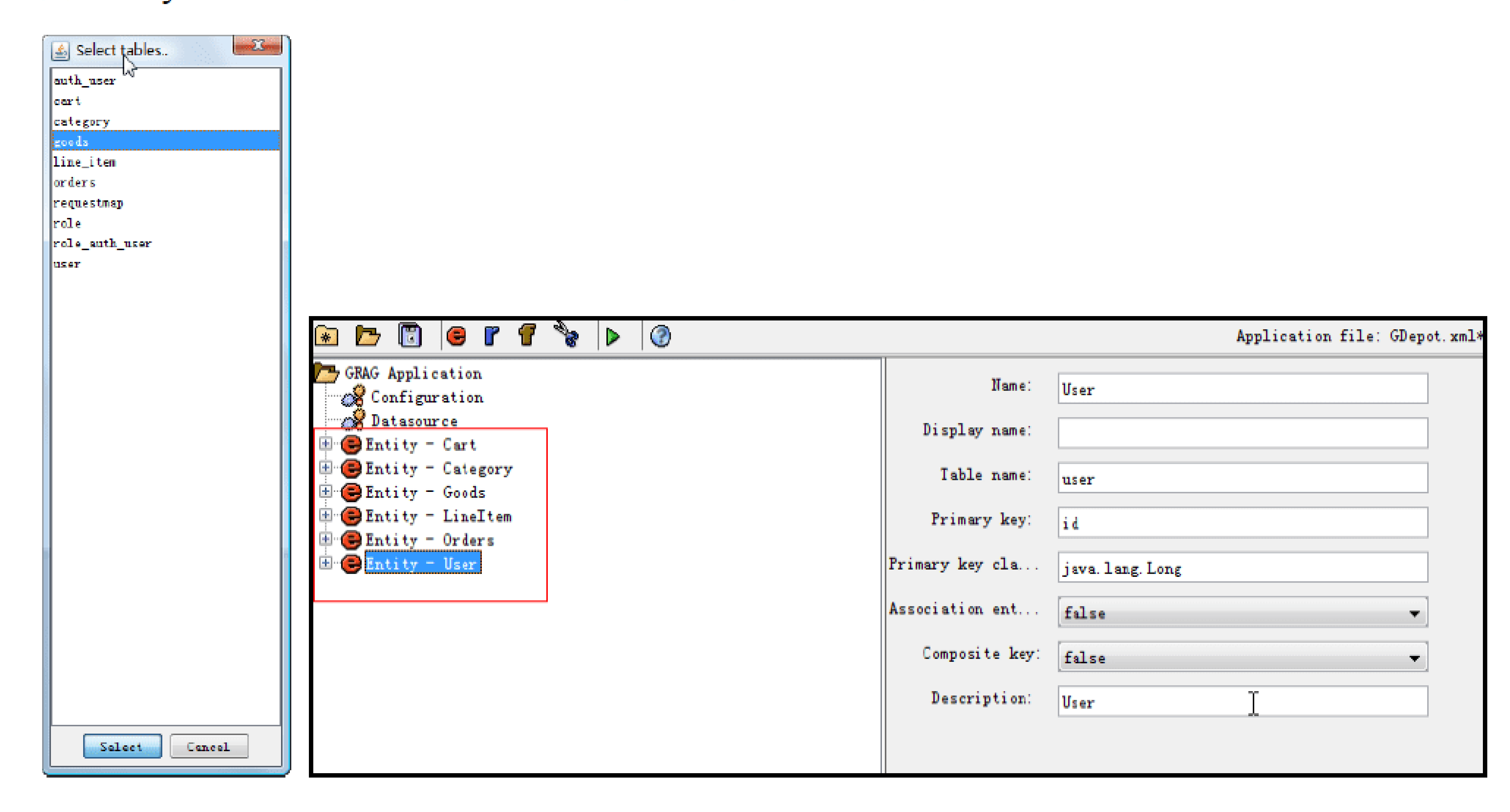


图 11-7　选择数据库表　　　　图 11-8　成功导入为实体类

当然，自动映射的类不可能完全满足需求，所以 GRAG 还提供了简单的修改功能。在左侧选择不同的实体类，然后就可以在右侧进行修改。修改完毕，执行 File|Generate Application 命令，或者单击工具栏的按钮，会将实体类导出成 Groovy 类文件。

生成的 Groovy 代码如下，以 Goods.groovy 为例：

```
/**
 * The Goods entity.
 *
 * @author
 * @version $Revision: $, $Date: 2006/07/07 20:04:13 $
 *
 *
 */
class Goods {
    static mapping = {
         table 'goods'
         version false
         id generator:auto, column:'id'
         version column:'version'
         description column:'description'
         photoUrl column:'photo_url'
         price column:'price'
```

```
            title column:'title'
            categoryIdCategory column:'category_id'
        }
        java.lang.Long id
        java.lang.Long version
        java.lang.String description
        java.lang.String photoUrl
        java.math.BigDecimal price
        java.lang.String title
        // Relation
        Category categoryIdCategory
        static constraints = {
            id(nullable: true, max: 19)
            version(nullable: false, max: 19)
            description(size: 1..255, blank: false)
            photoUrl(size: 1..255, blank: false)
            price(nullable: false)
            title(size: 1..255, blank: false)
            categoryIdCategory()
        }
        String toString() {
            return "${id}"
        }
    }
```

从上面的代码可以看出，GRAG 的功能很强大，它生成的 Domain 类中包括 mapping、属性、表间关系和 constraints。GRAG 确实能够减轻为兼容历史数据库时而手动编写映射代码的任务。但毕竟机器是无法取代人的，它自动生成的代码也并不完美：GRAG 没有利用默认的属性名与字段名的对应关系，而完全用 mapping{}进行配置。因此，它生成的 Domain 类会比较庞大。通常情况下，还需要对其生成的代码进行修改。毕竟 GRAG 现在还只是 1.0 版本，有理由相信它会不断进步的。

11.6 本章小结

本章对 GORM 的高级特性进行了比较完整的介绍。涉及到自定义映射、复杂的对象关联、数据库的查询、性能调优等。本章的最后一节介绍了 GRAG 工具，通过它可以帮助实现从数据库表结构生成 Domain 类。熟练掌握本章的内容，是灵活驾驭 Grails 的基础。

第 12 章 与 Spring 整合

通过在 GORM 中使用自定义映射，可以解决兼容遗留数据库的问题。如果遗留的产品中有封装的比较优秀的 Java 组件，也可以直接在 Grails 中使用。毕竟 Groovy 可以对 Java 程序进行透明调用。

此外，由于 Grails 是基于 Spring 的设计，使得很多外部的组件可以通过 Spring 引入系统。这样就可以使用 DI（DI，Dependence Injection，依赖注入）模式，设计出更加松耦合的系统。

本章将先对 Spring 进行简单的介绍，然后讨论 Grails 如何封装 Spring，并将重点介绍 Grails 提供的用于简化 Spring 配置文件编写的 DSL。本章的内容对于掌握 Grails 的 Web 开发，用处并不大，但如果希望学习 Grails 的原理，则有必要了解一二，不然在阅读 Grails 的源代码时可能会比较困惑。

12.1 依赖注入与 Spring 容器基础

12.1.1 依赖注入

一个设计优秀的系统应该遵循着“开闭原则”。所谓“开闭原则”，指的是一个软件实体应当对扩展开放，对修改关闭。也就是说，在加入新的功能时，应该避免修改现有部分。面向对象的设计方法中，会使用多态机制去实现开放。在 Java 世界中，进行系统设计时通常会先将业务逻辑抽象成一系列接口。显然，面向接口编程，系统的可扩展性更好，结构也更加清晰。

假设有接口 Service，和它的实现类 ServiceImp，其程序代码如下：

```
public interface Service {
    public void doSomething();
}

public class ServiceImp implement Service {
    public void doSomething() {
        //实现 doSomething 的业务
    }
```

```
    }
```

如果在 Client 类中调用 Service，则 Client 的程序如下：

```
public class Client {
    private Service service;
    public Client () {
        //使用 ServiceImp 的实例初始化 service
        service = new ServiceImp();
    }
}
```

此时，Client 类与 Service 接口的关系是关联关系，而与 ServiceImp 的关系是依赖关系。用 UML 表示如图 12-1 所示。

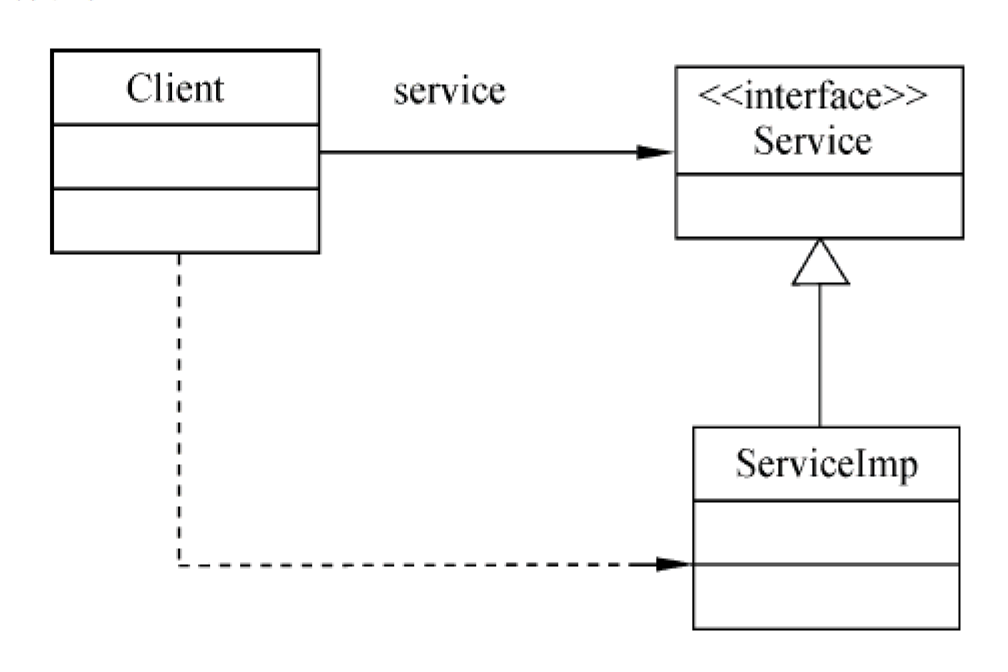


图 12-1　Client 与 ServiceImp 存在依赖关系

其中 Client 与 ServiceImp 的依赖关系这里并不喜欢。依赖关系的存在，就意味着在 ServiceImp 类不存在的时候，Client 类就无法编译通过；也意味着一旦想换用其他类实现 Service 接口，Client 类也需要进行修改。因此，不消除这个依赖关系，就没有真正地实现"开闭原则"。

使用依赖注入模式，也可称为控制反转（即 IoC），可以帮助实现消除上述的依赖关系。实现依赖注入，实际上就是使 Client 类拥有一个可写的 service 属性。在调用 Client 之前，再将 Service 子类的实例传入（注入）到 Client 中。

```
class Client {
    private Service service;
    public void setService(Service service) { this.service = service; }
    public Service getService() { return this.service; }
    ...
}
```

显然，此时的 Client 类与 ServiceImp 类不存在依赖关系。无论 Service 的实现类如何变化，Client 类都不需要修改，也不需要重新编译。此时的 UML 类图表示如图 12-2 所示。

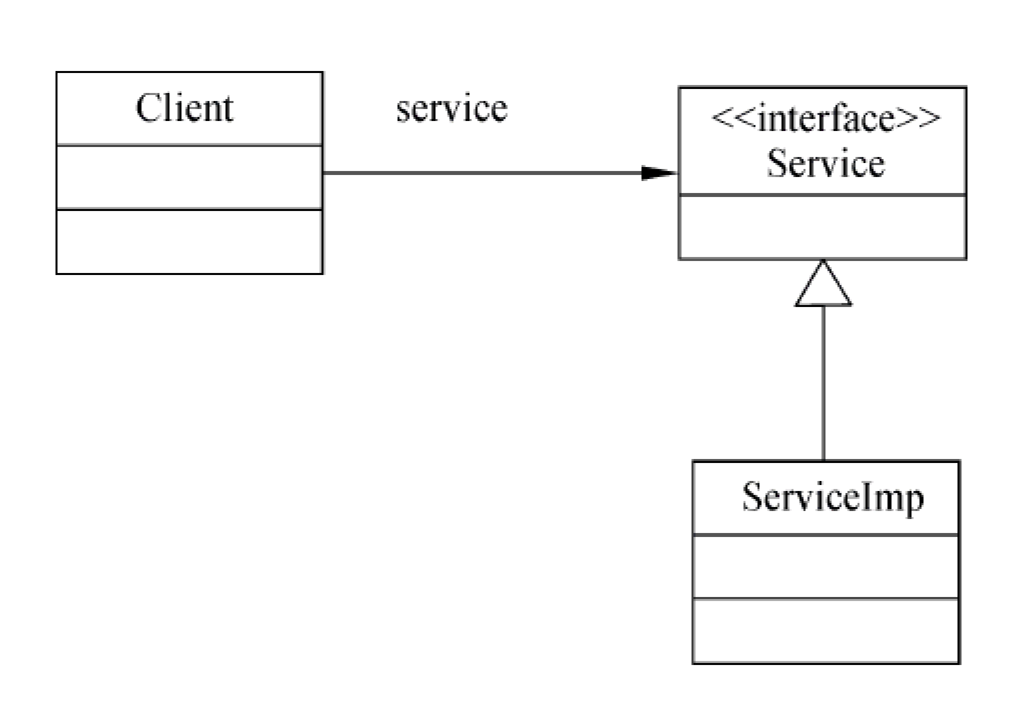


图 12-2　Client 与 ServiceImp 不存在依赖关系

凡事有利必有弊，虽然消除了依赖关系，但是 Client 的调用则变得更麻烦了：需要先使用 new 创建 Client 的实例，然后还需要将 Service 的实例传入 Client。如果忘了传，会有恼人的“空引用”异常抛出。

可以想象，当系统达到一定规模时，大量的接口与接口相互引用，UML 类图中会看到复杂的树状或网状关联结构。那么此时的初始化工作会变得异常复杂，消除依赖所带来的好处也几乎全被抵消了。

Spring 的诞生使得大量使用依赖注入成为了可能，因为它能够替程序员完成将依赖对象“注入”的任务，从而解决了初始化复杂的问题。

12.1.2　Spring 容器基础

Spring 设计了一个存放 Java 对象的容器，能通过 XML 或者 annotation 进行配置。容器内定义的对象可以自动或手动地配置依赖关系。由于 Spring 本身也是一个非常庞大复杂的 J2EE 框架，本书不会对其进行全面和深入的介绍，仅介绍一些必要的基础知识（基于 XML 的容器配置）。更多关于 Spring 的高级特性，读者可以参考 Spring 的官方教程[7]。

Spring 的核心就是它的 DI 容器，其 XML 配置文件内的主要内容就是配置容器内的 Java 对象。<bean></bean>用于在 Spring 容器内定义对象，例如：

```
<bean id="service" class="ServiceImp" />
<bean id="client" class="Client">
    <property name="service" ref="service"/>
</bean>
```

此时，如果通过容器去获取 client，就可以得到初始化了 service 属性的 client 实例，因为 Spring 会根据 client bean 的 property 的配置，引用容器中的 service 对象初始化 client 的 service 属性，即执行 client.setService(service)。可以通过 Spring 提供的 BeanFactroy 或者 ApplicationContext，根据 id 从容器中取出对象（不过一般不需要这样做，这里仅作为基本了解进行介绍）：

```
Client client = (Client) ctx.getBean("client");
```

<bean/>元素包含了大量可以配置的属性，可以实现对容器中的元素进行详细的配置，详细情况如表 12-1 所示。

表 12-1　Spring 配置 XML 中 bean 元素的常用属性

属性	功能	属性值说明	
id	定义对象在容器中的唯一名称。用于引用 bean	每个 bean 都需要有唯一的 id	
class	bean 对应的类	应该是包含完整的包名和类名的全称	
scope	bean 的作用域	singleton	默认的作用域，同一个 bean 在容器中创建唯一的实例
		prototype	每次访问都创建新的实例
		request	bean 的生命周期与 HTTP request 相同。每次 HTTP request 开始时，创建新的 bean。同一个 HTTP request 的多次访问，使用相同的实例
		session	bean 的生命周期与 HTTP session 相同。同一个 session 的多次访问，使用相同的实例
		global session	bean 的生命周期与 global HTTP session 相同
autowire	bean 属性自动绑定策略	no	不使用自动绑定
		byName	根据属性名和 bean id 实现自动绑定
		byType	根据属性类型和 bean 类型实现自动绑定
		constructor	根据构造函数进行初始化绑定
		autodetect	自动检测。会自动选择使用 constructor 方式，或者 byType 方式

说明：

（1）request、session 和 global session 都需要 Spring 运行于 Web 容器中，并且 Spring 能够访问 Web 容器的资源（例如 Grails 中 Spring 运行的方式）。

（2）使用 autowire 属性可以减少 XML 代码的体积。前面的实例可以简化为：

```
<bean id="service" class="ServiceImp" />
<bean id="client" class="Client" autowire="byName" />
```

但是使用 autowire 也有可能带来二义性的错误，例如，当容器中有多个实现了 Service 接口的类的 bean 时，若使用 byType 方式的 autowire，Spring 就会报错。

理解了 Spring 的 autowire 特性，就不难理解为什么在 Controller 中使用 Service 时无需对其进行初始化就可以直接调用：是 Spring 的 autowire 特性帮助完成了绑定。不过由于在 Controller 中定义 Service 时使用了无类型的“def”，Spring 只能通过 byName 的方式进行绑定。

12.2　在 Grails 中使用 Spring

Grails 是基于 Spring 的，当然也允许用户在 Spring 容器中添加其他的 bean。用户可以

在 config\spring 文件夹中添加 Spring 的配置文件，如 resources.xml，也可以是 resources.groovy。如果使用 XML 方式添加 bean，按上节介绍的方法在 resources.xml 中添加“<bean></bean>”即可。除此以外，Grails 中还提供了一种用于配置 Spring 的 DSL。使用 Spring DSL，配置过程更加简单，也更加灵活。

首先通过两段等价 XML 和 Spring DSL 进行对比，以理解 Spring DSL 的含义。

使用 XML：

```
<beans>
<bean id="service" class="ServiceImp" />
<bean id="client" class="Client">
    <property name="service" ref="service"/>
</bean>
</beans>
```

使用 DSL：

```
beans{
    service(ServiceImp)
    client(Client) {
        service = service
    }
}
```

如果需要修改 bean 的属性，可以按如下方式：

```
beans{
    service(ServiceImp)
    client(Client) {  bean ->
        bean.autowire = "byName"
    }
}
```

显然，使用 Spring DSL 更加简洁，而且可以实现动态配置 Spring。本质上，Grails 通过 grails.spring.BeanBuilder 实现 Spring DSL。Grails 在启动时，对 dataSource、Hibernate 等的配置和初始化，都是通过 Spring DSL 完成的。下面的代码摘录于 org.codehaus.groovy.grails.plugins.orm.hibernate.HibernateGrailsPlugin.groovy：

```
...
lobHandlerDetector(SpringLobHandlerDetectorFactoryBean) {
    dataSource = dataSource
}
...
```

无论是使用 XML 方式还是使用 Spring DSL 方式定义的 Spring 配置文件，所有配置的值都可以使用 10.2 节介绍的配置方法重新指定。其格式为：

```
[bean name].[property name] = [value]
```

在这时，定义 dataSource.driverClassName=com.mysql.jdbc.Driver，实际上是修改了 Spring 容器中定义的 dataSource bean 的 driverClassName 属性的值。

12.3 本章小结

Spring 框架是非常优秀的轻量级 J2EE 框架，正是由于它的简洁与优雅，Grails 选择 Spring 作为基础框架。本章对 DI（IoC）模式进行了简单的介绍，并以此引出了 Spring 框架的价值所在。本章对 Spring 的基础知识进行了简单的介绍，并且介绍了 Grails 提供的用于封装 Spring 配置信息的 DSL。相信读者在阅读了本章以后，能够配置 Spring 容器中的 bean，从而实现更加方便地调用遗留的 Java 组件。

第13章 深入 Controller

本质上，Grails 的 MVC 是基于 Spring MVC 的。相比 GORM，Controller 中新的知识点并不算多。本章将在对 Controller 的知识点进行简单小结之后，着重对 Web 流技术（Web Flow）进行介绍。

13.1 Controller 中常用的属性与方法

表 13-1、表 13-2 分别总结了 Grails Controller 中常见的属性与方法。

表 13-1 Controller 中常用的属性

属性名	作用
controllerName	取得当前 controller 的名字
actionName	取得当前 action 的名字
grailsApplication	org.codehaus.groovy.grails.commons.GrailsApplication 接口的实例，可以通过 grailsApplication 读取 Grails 的配置信息 config、元数据 metadata 以及类加载器 classLoader
request	Servlet API 中的 javax.servlet.http.HttpServletRequest 接口的实例。可参考 Sun J2EE 规范 API 文档： http://java.sun.com/j2ee/1.4/docs/api/javax/servlet/http/HttpServletRequest.html
response	Servlet API 中的 javax.servlet.http.HttpServletResponse 接口的实例。可参考 Sun J2EE 规范 API 文档： http://java.sun.com/j2ee/1.4/docs/api/javax/servlet/http/HttpServletResponse.html。 通过 response，可以实现输出二进制数据，通常用于输出图片、下载文件等
flash	flash 相当于一个 Map。它被自动存放到 session 中，并且自动在下次 request 执行完成时清空 Map 的内容。使用 flash，可以实现在两次 HTTP request 中共享数据，可以解决 request 对象中的数据在 redirect 之后会丢失的问题。通常用于向 GSP 页面传递文本消息
session	Servlet API 中的 javax.servlet.http.HttpSession 接口的实例。可用于在会话范围内共享数据
servletContext	Servlet API 中的 javax.servlet.ServletContext 接口实例。可参考 Sun J2EE 规范 API 文档： http://java.sun.com/j2ee/1.4/docs/api/javax/servlet/ServletContext.html。 可用于在全局范围内共享数据
params	params 相当于一个 Map，可以获取页面表单数据。 页面表单项的 name 和 value，提交后成为 params 的 key 和 value。 如果页面有多个表单项 name 相同（如复选框 checkbox），则提交后 params 可以通过 List 返回多个 value

表 13-2　Controller 中常用的方法

方法名	方法介绍
bindData	将 Map 的数据写入对象(Groovy 或 Java 对象均可),通常用于绑定 params 至 Domain。 用法：bindData(target, params, excludes, prefix) 参数： target: 目标对象，接收数据； params：数据来源，通常为 params 属性，也可是其他的 Map； excludes：可选参数，用于指定不希望被绑定的属性； prefix：可选参数，用于给指定 params 的 key 指定公共前缀，前缀与属性名用“.”分隔。使用前缀可以便于区分一次表单提交包含的多个对象的属性。 示例： bindData(goods,params) bindData(goods,params,["title","price"]) 不更新 goods 的 title 和 price 属性 bindData(goods, params, [],"goods") 假定 params 中的 key 包含 goods 前缀，例如：key 为"goods.title"
redirect	实现重定向（跳转），本质上是通过 HttpServletResponse 的 sendRedirect 方法实现的。 redirect 方法的参数是一个 Map，Map 中可以使用下列 key 配置跳转： controller：指定目标 controller； action：指定目标 action，如果没有指定 controller，则为当前 controller； id：用于向目标 action 传递的参数； uri：指定完整的跳转 uri，如/book/list，/book/get/1； url：完整的 Web URL，如 http://www.grails.org； params：使用 map 传递参数； fragment：指定 URL 中的锚点（/help.html#index 中#后面的部分），在使用 fragment 时不要包含“#”

render	输出文字、页面或者模板页面。render 方法的功能非常强大。
	例 1：输出文字（一般来说，用处不大）。
	render "Hello World"
	还可以指定 contentType 和编码
	render (text: "<xml>Hello World</xml>",
	　　contentType:"text/xml",encoding:"UTF-8")
	例 2：输出 GSP（最重要且最常用的用法）。
	使用 model map 向 GSP 传递数据
	render(view: "list", model : [goodsList: Goods.list()])
	
	若不指定 model，则自动将 controller 作为 model 传递给 GSP view，GSP 中可以直接访问 Controller 中定义的属性。（与 Webwork 或 Ruby on Rails 的传递方式相似）。
	render(view: "list")。
	例 3：输出 Builder 闭包。
	受益于 Groovy 强大的闭包和 MOP 机制，大量基于 Goovy 的 DSL 被开发出来（通常用于构造表示树形结构的数据）。
	
	通过 HTML Builder 输出 HTML
	render {
	　table (id: "news") {
	　　tr {
	　　td ("title")

续表

方法名	方法介绍

render	td ("date")
	}
	}
	}
	实际输出为：
	<table id="news"><tr><td>title</td><td>date</td></tr></table>
	如果指定了 render 的 contentType，还可以输出 XML 或 JSON 数据。
	
	XML 示例：
	render (contentType:"text/xml") {
	for(g in goodslist) {
	goods (title: g.title, price: g.price)
	}
	}
	实际输出为：
	<goods title='Grails' price='20.00'/><goods title='Groovy' price='20.00'/>
	
	JSON 示例：
	render (contentType:"text/json") {
	for(g in goodslist) {
	goods (title: g.title, price: g.price)
	}
	}
	实际输出为：
	{"goods":{"title":"Grails","price":20},"goods":{"title":"Groovy","price":20}}
	
	例 4：使用 convertor 输出。
	使用 convertor 可以简化 XML 或 JSON 的输出：
	import grails.converters.*
	…
	render Goods.get(params.id) as XML
	render Goods.list(params) as JSON
	
	例 5：使用 template 输出（在 GSP 中更常用）。
	render 方法的功能与 7.2 节介绍的 GSP render tag 相同，用法也相似。
	输出模板页面。
	如：render template: 'templateName'
	
	指定 model，模板页面可以通过 model 的 key 访问变量。
	如：render template: 'templateName', model:[var1:value1,var2:value]
	
	指定 bean（此时只传入一个对象，模板页面使用“it”访问 bean 的内容）。
	如：render template: 'templateName',bean:obj

续表

方法名	方法介绍
render	指定 bean，并且使用 var 指定页面访问 bean 的变量名称。 如：render template: 'templateName',bean:obj, var: 'goods' 此时模板页通过“goods”访问 obj。 指定 collection，则遍历集合时，每取出一个集合元素，将其传给模板，并输出一次 template。如果指定了 var，则模板中使用 var 指定的名称访问集合元素，否则使用 it。 如：render template: 'templateName',collection:Goods.list(), var: 'goods'

chain	实现多个 action 的链式调用，并维持每个 action 的 model 内容。本质上，chain 方法是用 flash 维持了 model 的内容，然后用 redirect 进行了跳转。
	例如：
	def first = {
	    chain(action:second,model:[one:1])
	}
	def second = {
	    chain(action:third,model:[two:2])
	}
	def third = {
	    [three:3])
	}
	如果访问 first action，最终的 model 为[one:1,two:2,three:3]。
	具体参数（Map 参数）：
	uri、action、controller、id、params：用法与 redirect 相同；
	model：要传递给下一个 action 的 model

withFormat	实现根据 HTTP 请求的 Accept 头信息或 URI 的扩展名，智能地输出不同格式的数据。
	例如：
	def list = {
	    def goods = Goods.list()
	    withFormat {
	        html bookList:books
	        js { render "alert('hello')" }
	        xml { render books as XML }
	    }
	}

13.2 自定义 URL Mapping

严格地讲，URL Mapping 并不是 Controller 中需要讨论的问题，但因为 Controller 与 URL 的关系是如此的紧密，所以将对自定义 URL 映射的问题，安排到了本节进行讨论。

第 2 单介绍过 Grails 的基础知识，URL 与 Controller 以及 action 的对应关系是：“/controllerName/actionName/id”。事实上，这个对应关系也是可以配置的，Grails 提供了自定义 URL Mapping 的方法：修改 grails-app\conf\UrlMappings.groovy，可以改变 URL 的映

射策略。默认的 UrlMappings.groovy 内容如下：

```
class UrlMappings {
    static mappings = {
      "/$controller/$action?/$id?"{
          constraints {
              // apply constraints here
          }
      }
      "500"(view:'/error')
    }
}
```

其中定义了两条策略："/$controller/$action?/$id?"表示解析 URL 中的 Controller、Action 和 id，然后执行相应的 Controller 和 Action 并传递 id 的值；而"500"(view:'/error')则表示，当执行状态码为“500”时，输出 error.gsp 页面。这里可以根据需要，自行配置 URL 的映射，例如定义：

```
"/book/$id" {
    controller = 'goods'
    action = 'show'
    type='book'
}
```

或者：

```
"/book/$id" (controller:'goods', action:'show', type:'book')
```

上面这两种写法是等价的，选用哪一种，完全取决于用户的喜好。此时，访问/book/1 与访问/goods/show/1?type=book 是等价的。

使用 URL Mappings 时，可能在 URL 的匹配上，存在一定的二义性。例如，很难说/book/1 指向的是 bookController 的默认 action，还是 goodsController 的 show action。URL 匹配的优先级不是哪个配置项更靠前，而是优先选择哪一个配置项更“特殊”。

/book/$id 中包含了固定的内容“book”，因而比"/$controller/$action?/$id?"更特殊，所以会优先匹配/book/$id。

下面再对默认的"/$controller/$action?/$id?"进行更深入的讨论，配置项中也可以定义与 Domain 相似的约束条件：

```
"/$controller/$action?/$id?"{
          constraints {
              // apply constraints here
          }
}
```

其中，controller、action 和 id 是预置变量，可以自动通过预置变量将匹配结果传递给 Controller、Action 和 id。action 和 id 后面的问号“?”表示这两个变量是可选变量，URL

中可以不包含该项内容。如果使用闭包方式定义映射，则还可以在闭包内定义 constraints 对 URL 的约束，例如，下面的映射限制 type 只能是 book 或者 food。通过约束条件，也可以使 URL 的映射变得更加特殊，从而实现优先使用该规则进行匹配：

```
"/$type/$id" {
    controller = 'goods'
    action = 'show'
    constraints {
        type(inList:['book', 'food']
    }
}
```

除了预置的变量，还可以自行定义其他的变量，例如，如下格式可以配置按日期显示 blog 文章列表：

```
/blog/$year?/$month?/$day? ( controller: 'blog', action: 'search' )
```

则请求/blog/2008/9/1，与请求/blog/search/?year=2008&month=9&day=1 是等价的，但显然第一个 URL 对用户更加友好。

进行 URL 映射时，还可以使用通配符。

（1）"/images/*.jpg"(controllers:"image")，表示所有的/images/*.jpg 都转到 image controller 中处理。

（2）"/images/**.jpg"(controllers:"image")，表示所有/images/下的任意子目录中的 jpg，都转到 image controller 中处理。

（3）"/images/$name*.jpg"(controllers:"image")和"/images/$name**.jpg"(controllers:"image")，表示转到 image controller 中处理，并且 jpg 的文件名会以 params.name 的形式传入控制器中。

正如默认的 URL Mapping 所示，可以对状态返回码进行映射：

```
"500"(view:"/errors/serverError")
"404"(view:"/errors/notFound")
"403"(view:"/errors/forbidden")
```

除此以外，还可以对 HTTP 请求的方法（method）进行映射：

```
"/product/$id"(controller:"product"){
    action = [GET:"show", PUT:"update", DELETE:"delete", POST:"save"]
}
```

使用了 URL Mapping，无疑可以给用户提供更加友好的 URL。但是，是否给用户在页面上输出链接带来麻烦呢？答案是不会。g:link 标签已经内置了支持 URL 自动改写的方法。仍以 blog 的 URL 为例，如果定义了 blog URL 映射：

```
/blog/$year?/$month?/$day? ( controller: 'blog', action: 'search' )
```

则使用如下方法构造链接：

```
<g:link controller="blog" action="show" params="[year:2008, month:9,
day:1]">
Article List
</g:link>
```

这将输出：

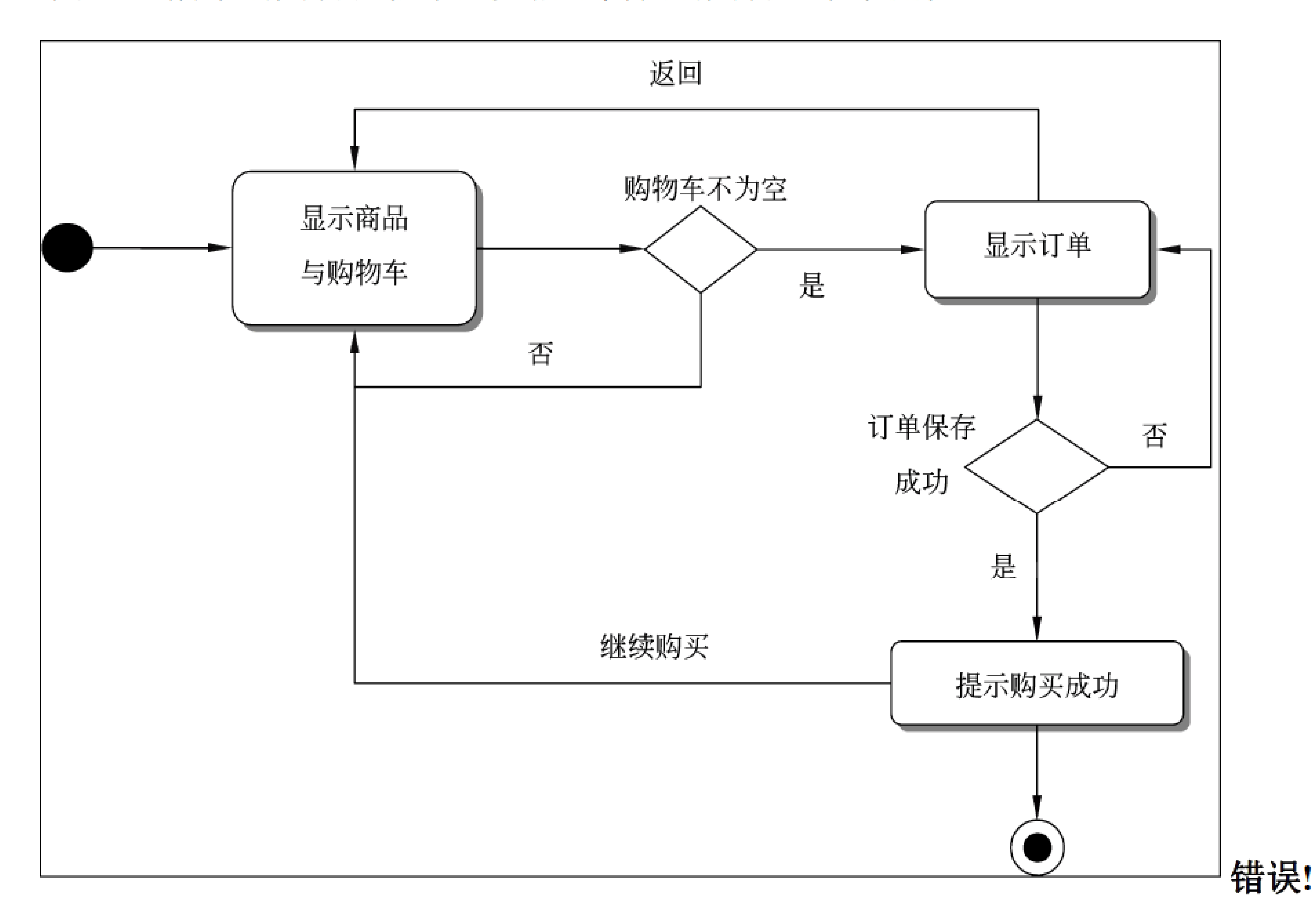
```
<a href="/blog/2008/9/1">Article List</a>
```

怎么样？是不是很简单？

13.3 Web Flow

Grails 从 1.0 开始就支持 Web Flow 的概念。它的底层是依靠 Spring Web Flow 项目实现的[1]。使用 Web Flow，可以轻松地编写有状态和基于会话的 Web 应用。通过 Web Flow，可以实现跨越多个 HTTP 请求保持状态。Web Flow 应用可以将业务理解为一个有穷状态机，不同的事件会触发业务状态变化，Web 页面也相应地从一种状态变为另一种状态。

图 13-1 所示的购物流程中主要描述简化的购物、下单流程。



错误!

图 13-1 简化的购物车工作流程

购物过程的实际操作要比图 13-1 中画的更复杂些，需要包含将商品添加到购物车、将

[1] Spring Web Flow 是 Spring 框架的一个子项目。参考 http://www.springframework.org/webflow 了解详细内容。当前版本的 Grails 使用的是 Spring Web Flow 2.0m1 的版本，这并不是当前最稳定的版本，因此希望读者自己决定是否使用 Web Flow。

购物车中商品删除、修改购物车中商品数量等内部操作。如果将整个购物过程看成是一个 flow，可以将显示“商品与购物车”、“显示订单”等看成是这个 flow 中的几个状态。用户的操作以及业务的约束等，将触发应用在不同的状态间相互转换。

Spring Web Flow 就是针对这样一类需求而诞生的，它通过 Flow 的概念直观地将业务上的流程在 Web 应用中体现出来。Grails Web Flow 更将 Spring Web Flow 的技术进行封装和简化，提供一种针对 Web Flow 的 DSL，因而无需再编写繁琐的基于 XML 的 Flow 配置程序。下面将揭开 Grails Web Flow 的神秘面纱。

首先，创建 Flow。在普通的控制器中创建名称以“Flow”结尾的闭包，例如：

```
class GoodsController{
    def orderGoodsFlow = {

    }
}
```

如果闭包名称是以“Flow”结尾，则 Grails 不再把它当作 action 处理。在创建 Flow 后，就可以在 Flow 中定义状态。Flow 中的一个节点表示 Flow 的一个状态。而 Flow 的第一个状态，是 Flow 执行的入口，称为初始状态。如果 Flow 中的某些状态使用 redirect 方法跳出了 Flow，或者不能再转入其他状态，这样的状态称为结束状态。当 Flow 进入结束状态后，只能重新进入 Flow 的初始状态，而不能再进入 Flow 的其他状态。

Flow 中的状态可分为两类：视图状态（View State）和操作状态（Action State）。下面分别进行介绍。

视图状态：视图状态用于输出 Web 页面，在视图状态中不能执行 redirect，也不能定义 action 节点。视图状态中不能直接编写与 Web Flow 状态变化无关的代码，如查询数据库，但可以在事件处理时编写。

操作状态：用于执行“操作”，如查询、更新数据库、处理表单等。操作状态中通过 action 节点定义操作。

为了便于理解，这里在 orderGoods 中开发几个状态：

```
def orderGoodsFlow = {
    listGoods {
        action{
            [cart: orderService.prepareCart(session),
            goodsList:
            new grails.orm.PagedResultList( Goods.list( params ) ,
            Goods.count() ) ]
        }
        on('success'). to 'displayGoodsList'
        on(Exception). to 'handleException'
    }
    displayGoodsList {
        on('addGoodsToCart').to 'addGoodsToCart'
        on('order'){
```

```
        }.to 'displayOrder'
    }
    addGoodsToCart{
        action{...}
        on('success').to 'listGoods'
        on(Exception). to 'handleException'
    }
    dispalyOrder { ... }
}
```

其中 listGoods 是 orderGoodsFlow 的初始状态。通过浏览器访问 goods/orderGoods，则自动进入 listGoods 状态。

listGoods 状态是一个操作状态，进入 listGoods 状态，会自动执行 action 节点中的程序，从而完成读取购物车和商品列表的操作。当 action 节点中的程序执行正确无误时，会自动触发 success 事件，并转换当前状态为 displayGoodsList 状态；而当 action 节点中的程序抛出异常时，on(Exception)会被触发从而跳转到异常处理状态（页面）。

displayGoodsList 状态是一个视图状态。默认情况下，进入视图状态，会输出 GSP 页面（/grails-app/views/控制器名/Flow 名/状态名.gsp）。对于 displayGoodsList，会输出/grails-app/views/goods/orders/displayGoodsList.gsp 页面的内容。当然，这里也可以使用 render 方法输出其他页面，例如：

```
render(view: '/goods/list.gsp')
```

在使用 Web Flow 时，作用域的使用变得很重要。listGoods 状态的 action 节点的返回值会随着状态变化的网络传播到下一个状态中。于是 displayGoodsList 对应的 GSP 页面可以访问 cart 和 goodsList。

当 Flow 的执行进入视图状态，就需要靠用户的操作去触发状态变化了。用户可以通过单击按钮或链接实现触发事件。

通过按钮（本质上是表单中加 g:submit）：

```
<g:form action="orders">
   <input type="hidden" name="goodsId" value="${goods.id}" />
<g:submitButton name="addGoodsToCart"
        value="Add To Cart"></g:submitButton>
</g:form>
```

其中，表单（g:form）的 action 属性决定 Web Flow 的名称；g:submitButton 的 name 属性决定 Web Flow 被触发的事件名称。并且，在表单中，仍然可以提交其他数据，以添加商品到购物车为例，显然需要提交商品的 id。

```
addGoodsToCart{
    action{
        if(session.userId && params.goodsId) {
            orderService.addToCart(session,params.goodsId)
```

```
        }
    }
    on('success').to 'listGoods'
    on(Exception). to 'handleException'
}
```

不难想象，在操作状态的 action 节点中也是通过 params 接收表单参数的。

对于下单事件，可以由单击链接所触发：

```
<g:link action="orders" event="order" >下单</g:link>
```

与 g:form 的使用相似，g:link 的 action 属性此时用于指定 Web Flow 的名称。但不同的是事件的名称是通过 event 属性指定的。根据图 13-1所示可知，在转换到显示订单状态前需要对购物车的内容进行检查，只有购物车内容不为空时才允许进行状态转换。有两种实现办法，一种是先进入一个操作状态，在这个状态中对表单进行检查，并根据检查结果进入下一步的状态，例如：

```
checkCartNotEmpty{
    action{
    def cart = orderService.prepareCart(session)
        if( cart?.lineItems?.size())
            yes()
        else
            no()
    }
    on('yes'). to 'displayOrder'
    on('no'). to 'listGoods'
}
```

这里的 yes()和 no()方法触发了 yes 或 no 事件[1]，与画流程图分支逻辑时使用的“是”、“否”含义一致。另一种办法是直接在事件触发时执行检查操作：

```
on('order'){
    def cart = orderService.prepareCart(session)
    if(! cart?.lineItems?.size())
        error()
}.to 'displayOrder'
```

注意，在检查发现购物车内没有商品时，使用 error()触发了 error 事件。于是当前的状态转换不会再被执行。上面的这种做法对于进行表单验证之类的操作显得十分方便。

接下来，深入讨论一下 Web Flow 中作用域的问题。首先需要明确一点，Web Flow 中的每次状态转换都执行了不同的 HTTP 请求。因此，不同的状态之间不能再用 request 共享数据。在 Web Flow 中，可以使用 flow 作用域或者 conversation 作用域。flow 作用域可以

[1] 从 Grails 的源代码中可以看到，如果在 action 节点中执行一个没有定义过的且无参数的方法，该方法就被理解为触发了事件。

在一个 Web Flow 的众多状态之间共享数据，而 conversation 作用域还要更大一些，可以在主 Web Flow 和子 Web Flow 间共享数据（子 Web Flow 将会在后面介绍），例如：

```
flow.cart = orderService.prepareCart(session)
conversation.order = new Order()
```

视图状态在调用 GSP 的时候，会自动将 flow 和 conversation 中的数据合并到 model 中，因而 GSP 页面可以直接访问 flow 和 conversation 中的数据。

本质上讲，flow 和 conversation 中的数据不是通过 HTTP session 维护的，而是通过序列化的表单。因而只有实现了 Serializable 接口的类，其实例才能存放到 flow 和 conversation 中，否则执行时会报错：

```
class Goods implements Serializable {
...
}
```

对于操作状态，action 节点的返回值是通过 flow 维护的，因而 action 也只能返回实现了 Serializable 接口的对象。

当业务非常复杂时，希望将其划分成多个子流程。每个子流程可以理解为全局主流程的一个新状态，如图 13-2 所示。

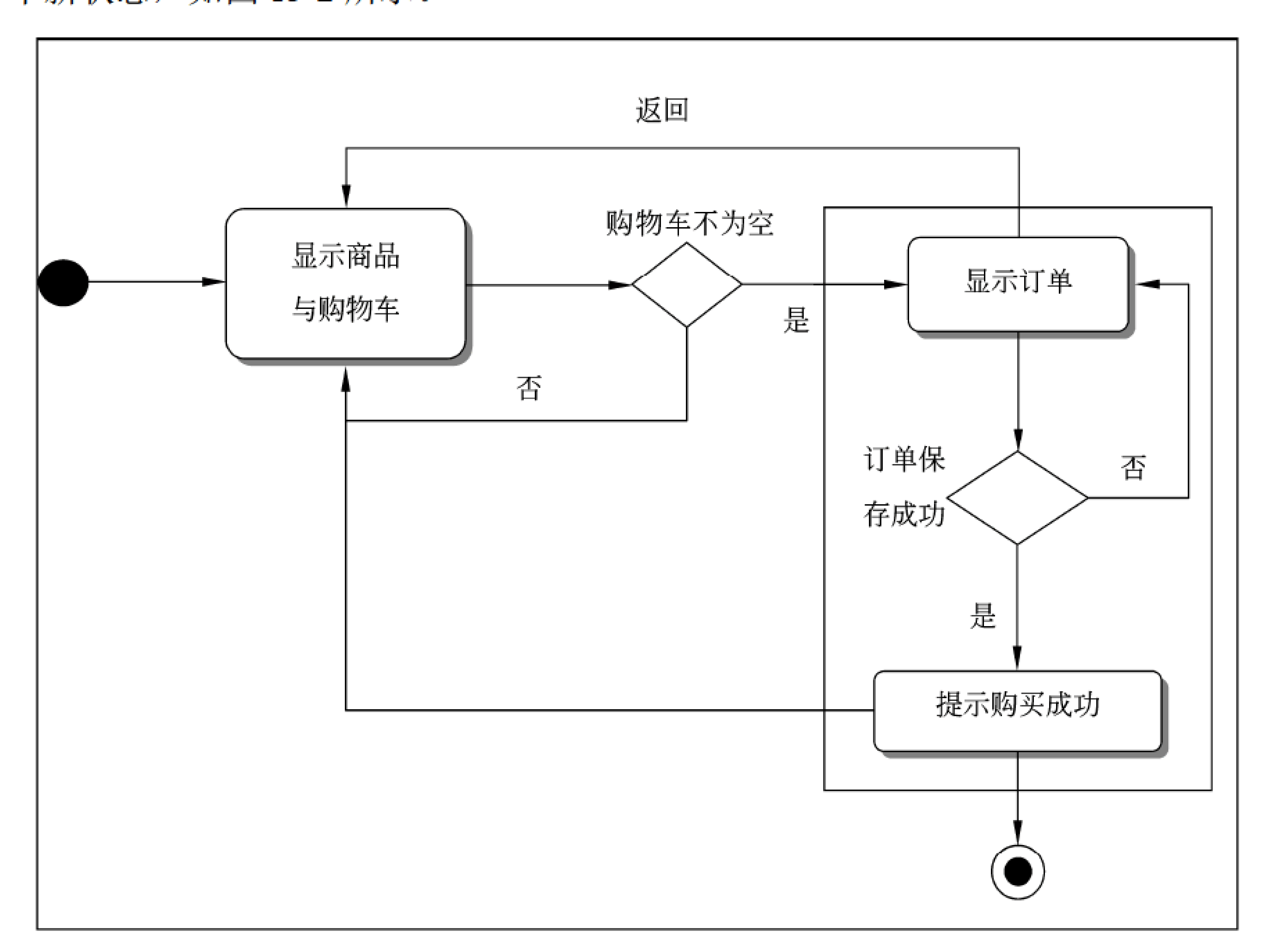


图 13-2　将右侧方框中的几个状态定义为一个子流程

Spring Web Flow 中通过“子 Web Flow”的概念，可以支持这样的子流程。以提交订单的过程为例，将提交订单的几个状态封装成一个子 Web Flow。

```
def orderSubmitFlow {
```

```
        displayOrder {
            on('save').to 'save'
            on('return').to 'listGoods'
        }
        save{
            action {
            //编写订单保存逻辑
            }
            on('success').to 'success'
            on('error').to 'displayOrder'
            on('exception').to 'displayOrder'
        }
        success{
            on('exit').to 'exit'
            on('continue').to 'listGoods'
        }
        listGoods()
        exit()
    }
```

orderSubmitFlow 中包含了 5 个状态。因为 listGoods 状态和 exit 状态不能再转换为其他状态，所以它们是 orderSubmitFlow 的结束状态。主 Web Flow 中的视图状态可以使用 subflow 方法引用子 Web Flow。而子 Web Flow 的结束状态可以成为主 Web Flow 中触发状态转换的事件：

```
    def ordersFlow {
        ...
        displayOrder {
            subflow(orderSubmitFlow)
            on('exit').to 'logout'
            on('listGoods').to 'listGoods'
    }
        ...
    }
```

因为主 Web Flow 和子 Web Flow 是不同的 Web Flow，所以无法通过 flow 作用域共享数据。需要使用 conversation 作用域，以实现在主 Web Flow 和子 Web Flow 间共享数据：

```
    converstation.cart = flow.cart
    //此时可以通过 converstation 将 cart 传递给子 Web Flow
```

13.4　本章小结

本章对 Controller 和 URL Mapping 进行了总结性的介绍，并重点补充了前面章节没有

介绍过的 Web Flow 的内容。不过在这里提醒一下读者，使用 Web Flow 的前提是应用比较复杂，而且有严格的“流程”要求。否则，使用 Web Flow 不但没有优势，还会带来更多的麻烦。另一方面，由于 Spring Web Flow 本身也是一个非常强大且完整的框架，Grails 并没有封装它的全部功能，再加上相关文档比较匮乏，因此建议读者一定要在确定使用 Web Flow 的功能前，先参考一下 Spring Web Flow 的文档。有了这些准备，才能真正驾驭 Grails Web Flow 并发挥出它的真正威力。

第14章 深入 Groovy Server Page

GSP (Groovy Server Page)是 Grails 视图的默认输出方式。这是一种使用方式与 JSP 相似，但更加灵活的页面视图技术。相信读者读到这一部分的时候，应该已经对 GSP 不再陌生了，本章就将对前面学习过的 GSP 技术进行简单的小结，然后重点介绍自定义 GSP 标签的创建方法。

14.1 GSP 基础知识

14.1.1 GSP 输出表达式

GSP 的视图技术与 JSP 是非常相似的，首先 JSP 可以在页面中编写 Java 代码，而 GSP 可以在页面中编写 Groovy 代码。JSP 与 GSP 代码都可以写在“<% %>”中。

```
<%
if( goods) {
        out.print(goods.title)
        out << goods.price // “<<” 是 GSP 中为 out 重载的运算符，
                             //与 print 方法含义相同
%>
    <br/>
<%
    }
%>
```

其次在 GSP 和 JSP 中，都可以用“<%= %>”输出表达式的值。

```
<%= goods.title %> 等价于 <% out << goods.title %>
```

GSP 也支持部分的 JSP 风格的页面指令，如：

```
<%@ page import="java.awt.*" %>
<%@ page contentType="text/json" %>
```

然而，上面的方法仅仅是为了使熟悉 JSP 的人能够尽快上手，并不是 GSP 中推荐作法。GSP 推荐用户使用 GString 风格（${ expr }）表达式输出方法：

```
${goods.title} 与 <%= goods.title %>等价
```

14.1.2 GSP 中预定义的变量与作用域

GSP 中预定义的变量与作用域如表 14-1 所示。

表 14-1 GSP 中预定义的变量与作用域

变量与作用域	简介
application	javax.servlet.ServletContext 实例。可参考： http://java.sun.com/j2ee/1.4/docs/api/javax/servlet/ServletContext.html
applicationContext	Spring 容器： org.springframework.context.ApplicationContext 的实例。可以通过 applicationContext 取得容器内的 bean。可参考： http://static.springframework.org/spring/docs/2.5.x/api/org/springframework/context/ApplicationContext.html
out	输出流。用于输出页面
flash	与 Controller 中的同名属性相同
grailsApplication	
out	
params	
request	
response	
session	
webRequest	封装了 Grails 的 Web 请求，可参考： http://grails.org/doc/1.0.x/api/org/codehaus/groovy/grails/web/servlet/mvc/GrailsWebRequest.html。 webRequest 中包含了大量的实用属性： webRequest.actionName 返回 action 名称 webRequest.controllerName 返回 controller 名称 webRequest.applicationContext 同 applictionContext webRequest.currentRequest 同 request webRequest.currentResponse 同 response webRequest.parameterMap 同 params webRequest.flashScope 同 flash webRequest.out 同 out webRequest.isFlowRequest() 判断当前请求是否为 flow 请求

14.2 GSP 标签库

曾经有不少人以标签库（taglib）为借口诟病 GSP，乃至攻击 Grails。其实更多的原因是他们不了解 GSP 的 taglib，只是想当然地把他们已有的 JSP 的知识套用在 GSP 上。事实上，GSP 中的 taglib 更加灵活，也更加简单。

GSP 中的标签调用方式极为灵活：除了可以按标签的方式调用外，还可以按函数的方式进行调用。对于引发命名冲突的标签，可以调用时加上前缀，如：

```
<g:message code="goods.title" />
${message( code: "goods.title" )}
${g.message( code: "goods.title" )}
```


三者完全等价。

因此，对于不喜欢标签的用户，完全可以按照函数的方式调用标签。此时，GSP 与 Ruby on Rails 的 rhtml 就十分相似了。

14.2.1　常用的内置标签

借用 JSP 标签的概念，将标签<xxx>…</xxx>中间的部分称为 body。GSP 的常用内置标签如表 14-2 所示。

表 14-2　常用内置标签

分类	标签	用法
逻辑判断	if elseif else	语义上与 Groovy 的 if…else 完全一致。 当 test 属性为真时[1]，输出 body；否则输出<g:else>的 body <g: if test="${session.user?.type==1}">…</g:if> <g:elseif test="${session.user?.type==2}">…</g: elseif > <g:else>…</g:else> 其中，elseif 或 else 必需与 if 联用
迭代	each	遍历集合: <g:each in="${[1,2,3]}" var="num"> ${num} </g:each> 详见第 4 章
查找	findAll	只遍历集合 expr 为真的子集。相当于 <g:each in="${books}" var="book"> <g:if test="${expr}"> … </g:if> </g:each> 例如： <g:findAll in="${books}" expr="it.author == 'Stephen King'"> Title: ${it.title} </g:findAll>

续表

<table>
<tr><th>分类</th><th>标签</th><th>用法</th></tr>
<tr><td></td><td>grep</td><td>对 in 属性对应的表达式执行 grep 方法后返回的集合进行遍历。
例如：
<g:grep in="${users}" filter="Student.class">… </g:grep>
等价于 Groovy 代码：
users.grep(Student.class).each({ … })

filter 的内容还可以是正则表达式：
<g:grep in="${books*.title}" filter="/(grails)|(groovy)">… </g:grep></td></tr>
<tr><td>循环</td><td>while</td><td>与 Groovy 中 while 循环的语义相同。test 属性为真时，执行循环
<g:while test="${ i< 5}">
${i++}
</g:while></td></tr>
<tr><td>链接</td><td>link</td><td>参考第 4 章，输出链接:
<g:link controller="book" action="show" id="1">查看</g:link>
返回：
查看</td></tr>
<tr><td></td><td>createLink</td><td>与 link 标签使用相似，但只返回 link 标签的 href 属性内容。
<g:createLink controller="book" action="show" id="1"/>
返回：
/book/show/1</td></tr>
<tr><td></td><td>createLinkTo</td><td>与 createLink 标签功能相似，通常用于寻找静态资源。dir 属性指定目录，file 属性指定文件名：
<g:createLinkTo dir="css" file="main.css" />
返回:
/appName/css/main.css
其中 appName 指工程名</td></tr>
<tr><td>变量赋值</td><td>set</td><td>为变量赋值。
<g:set var="bookName" value="grails" scope="session"/>
或
<g:set var="bookName" scope="session">grails</g:set>
等价于<% session.bookName = "grails" %>
其中的 scope，如果不指定，则默认为当前页面中的变量。
一般来说，更倾向于使用 Groovy 代码（<% %>）为变量赋值</td></tr>
<tr><td>输出模板</td><td>render</td><td>参考 Controller 中的 render 方法</td></tr>
<tr><td>错误相关</td><td>hasErrors</td><td>判断 Domain 对象是否包含校验错误信息，为真时输出 body。参考第 5.1 节。
<g:hasError bean="goods" field="price" >…</g:hasError></td></tr>
<tr><td></td><td>renderErrors</td><td>输出 Domain 对象的全部错误信息。参考第 5.1 节。
<g:hasError bean="${goods}" >
    <g:renderErrors bean=="${goods}" />
</g:hasError></td></tr>
</table>

[1] 这里的真与假，指的是 Groovy 概念上的真假，而非 Java 概念的真假。

续表

分类	标签	用法
	eachError	遍历 Domain 的错误信息。参考第 5.1 节。 <g:hasError bean="${goods}" > <g:eachError bean=="${goods}" > <g:message error="${it}"/> </g:eachError> </g:hasError>
格式化输出	formatDate	使用 java.text.SimpleDateFormat 格式化输出 java.util.Date 类型的日期。例如： <g:formatDate format="yyyy-MM-dd" date="${date}"/>
	formatNumber	使用 java.text.DecimalFormat 格式化输出数字。例如： <g:formatNumber number="${number}" format="\\$###,##0" />
i18n	message	输出 i18n 资源文件中定义的文本。 error：将 error 对象转换为消息文本，用于输出错误信息； code：根据资源文件中的 key 输出文本，用于输出自定义 i18n 文本； args：向资源文本传递参数，可参考第 5.1 节； encodeAs: 对输出文本进行编码处理，如 HTML、JavaScript、URL 等，也可以使用用户自定义的 Codec，例如第 5 章定义的 Password； default：默认消息，当资源文件中找不到 error 或 code 对应的文本时，则输出默认消息。 例如： <g:message code="goods.price" encode="HTML" />
分页与表头	sortableColumn	输出表头，包含控制排序的链接。参考第 6.3 节。 <g:sortableColumn property="releaseDate" defaultOrder="desc" title="Release Date" />
	paginate	输出分页导航链接。参考第 5.3 节。 <g:paginate controller="goods" total="${Goods.count()}" />
Layout 相关	layoutHead	输出原始页<head></head>之间的内容
	layoutTitle	输出原始页<title></ title >之间的内容
	layoutBody	输出原始页<body></body>之间的内容
	pageProperty	输出原始页某一元素的属性值 <g:pageProperty name="meta.menu" />
表单	form	输出<form></form>。可指定 action、controller、uri，如： <g:form action="save" controller="goods"> … </g:form>
	select	输出下拉列表框。可参考第 4.3 节。 from：非空，选项集合； value：可选，默认选中的选项； optionKey：可选，设置选项的 key； optionValue：可选，设置选项的值； noSelection：可选，用于指定默认选中的选项。

续表

分类	标签	用法
	select	例如： <g:select **optionKey="id" optionValue="categoryName"** from="${Category.list()}" name="category.id" value="${goods?.category?.id}" > </g:select> 其中 optionKey 设置了 option 的 value 属性，optionValue 设置了 option 的值。 <option value="${category.id}"> ${category.cagegoryName} </option>
	actionSubmit	输出提交按钮，可以指定接收请求的 action。 <g:actionSubmit value="更新" action="Update" />
	datePicker	下拉列表框选取时间日期。 name：非空，表单项的名称； value：可选，默认选中的时间，如果不指定，则为当前时间； precision：可选，可选的时间精度，包含'year'、'month'、'day'、'hour'、或者'minute'，默认为'minute'； noSelection：可选，当没有指定可选值，默认选中显示的内容（与 select 标签使用相同），例如 ['':'-Choose-']； years：可选的年的范围，可以是 Groovy 的 range，如${2001..2008}，也可以是 List，如${[2005, 2006, 2008]}

Grails 中还内置了部分用于 Ajax 开发的标签，详见第 14.3 节。

14.2.2　开发自定义标签

回顾了 GSP 中常见的标签后，下面介绍如何开发自定义的标签。GSP 的 taglib 与 JSP 相似，但远比 JSP 要简单与灵活，它甚至支持在运行时重新载入更新。开发自定义标签首先需要创建 taglib 类。可以在控制台执行命令 create-tag-lib 创建 taglib 类：

```
>grails create-tag-lib simple
Welcome to Grails 1.0.4 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: D:\grails-1.0.4

Base Directory: D:\workspace\GDepot
Note: No plugin scripts found
Running script D:\grails-1.0.4\scripts\CreateTagLib.groovy
Environment set to development
     [copy] Copying 1 file to D:\workspace\GDepot\grails-app\taglib
Created TagLib for Simple
```

```
    [copy] Copying 1 file to D:\workspace\GDepot\test\integration
Created TagLibTests for Simple
```

执行上面的命令，Grails 会在 grails-app\taglib 文件夹下创建名为 SimpleTagLib 的 Groovy 类。当然，用户也可以自己手动创建类似的 Groovy 类，不过类名一定要以 Taglib 结尾。创建了 TagLib 类，接下来就可以开发具体的标签了。标签类中可以访问的预定义属性包括：actionName、controllerName、flash、params、request、response、servletContext、session。用法与 GSP 中一致，这里不再赘述。

在 TagLib 类中添加如下的闭包：

```
class SimpleTagLib {
    def greet = { attrs, body ->
        out << "Hello"
    }
}
```

于是，可以在 GSP 页面上编写<g:greet />或者${greet()}调用该标签。上面的例子虽然简单，也能说明以下一些问题。

（1）TagLib 类中的每个闭包对应为一个具体的标签。

（2）在 GSP 页面中使用标签，无需编写配置文件，更无需添加形如<%@ taglib prefix=…的页面指令。

（3）out 对象的本质是文本流，但并不是 HttpServletResponse 的输出流。如果以调用方法的方式调用标签，函数的返回值就是 out 的内容。因此<% greet() %>不会输出“hello”，而<%= greet() %>或者${greet()}会输出“hello”。

接下来，开发更加复杂的标签。

首先，开发带属性的标签：<g:greet name="XXX" />。

细心的读者可能会注意到，标签闭包有两个参数：第一个参数 attrs（Map 类型）是标签的属性集合；第二个参数 body（闭包类型），执行 body()可以得到标签 body 的内容。通过“attrs.属性名”，可以访问到页面给标签传递的属性值：

```
def greet = { attrs, body ->
    out << "Hello "
    if(attrs.name) {
        out << attrs.name //使用 attrs.name 访问标签传入的 name 属性
    }
}
```

于是，在页面上调用<g:greet name="Matt" />或者${g.greet(name: "Matt")}可以得到输出：

```
hello Matt
```

十分神奇的是，name 属性不仅支持静态文本（Matt），也支持动态的内容，如：

```
<g:greet name="${goods.title}" />
```

而读者无需去关心究竟 goods.title 的值是如何取出的。

下一步，开发支持 body 的标签，如：<g:greet>Matt</g:greet>。正如前面所述，只需要执行 body()，就可以得到标签 body 的内容。

```
def greet = { attrs, body ->
    out << "Hello "
    out << body()
}
```

<g:greet>Matt</g:greet>将输出 Hello Matt。千万不要小看 body 闭包的威力，虽然只是简单地进行一下调用，但可以支持无限复杂的标签内容的解析执行，甚至包括嵌套标签的执行：

```
<g:greet><g:greet>${goods.title}</g:greet></g:greet>
```

假设 goods.title 的内容为 Groovy，则上面的标签将输出：

```
hello hello Groovy
```

某些情况下，标签需要向其 body 传递参数。例如 each 标签，它需要将每次遍历取出的对象传递给 body 内容。传递的方法十分简单，在调用 body 闭包时传入参数即可。

下面举例说明问题，设计一个执行循环的标签，包含 3 个属性：初始值 start、结束值 end、单步值 step。假设当前只支持整数循环，实现代码如下：

```
def forLoop = { attrs, body ->
    def start = attrs.start.toInteger()
    def end = attrs.end.toInteger()
    def i = start
    def step = 1
    if(attrs.step) step = attrs.step.toInteger()
    while(i in start..end) {
        out << body(i)
        i += step
    }
}
```

上面的代码为了简单，没有做有效性检查，正常情况需要先判断 start 和 end 是否为 null。其中的 body(i)实现了将数据传入标签的 body 中。但是，由于没有指定数据在 body 中的名称，此时 body 中的代码只能使用“it”访问传入的数据：

```
<g:forLoop start="0" end="10"><p>${it}</p></g:forLoop>
```

模仿 Grails 提供的 each 标签，这里也加入 var 属性，使用户可以自己定义 body 中使用的变量名：

```
def forLoop = { attrs, body ->
    def start = attrs.start.toInteger()
    def end = attrs.end.toInteger()
    def var = attrs.var ? attrs.var : "index"
    def i = start
    def step = 1
    if(attrs.step) step = attrs.step.toInteger()
    while(i in start..end) {
        out << body([(var):i])
        i += step
    }
}
```

此时，调用 body 闭包时，需要传入 Map，Map 的 key 决定 body 代码访问传入数据的变量名，而 Map 的 value 决定了传入数据的内容。这里应该注意一点，Map 必须写成[(var):i]而不是[var:i]，因为[var:i]永远表示 Map 的 key 是字符串“var”。

此时的 forLoop 标签调用方法如下：

```
<g:forLoop start="0" end="10" var="i"><p>${i}</p></g:forLoop>
```

读者会注意到，自定义的标签在默认情况下也被放到了 Grails 默认的命名空间（namespace）“g”中。事实上，可以自己定义标签的命名空间，只需要在标签类中定义静态属性 namespace：

```
class SimpleTagLib {
static namespace = "my"
    ...
}
```

此时调用标签的写法为：

```
<my:greet />
```

和

```
<my:forLoop start="0" end="10" var="i"><p>${i}</p></my:forLoop>
```

14.3 Grails 对 Ajax 的支持

默认情况下，Grails 是通过 JavaScript 的 framework“Prototype”对 Ajax 提供了支持。它提供了一些标签，可以简化开发 Ajax 应用的任务。Grails 标签简化开发的核心思想就是：后台提交一个请求（HttpXmlRequest），当请求完成时，更新页面上的一部分内容。

Grails 的标签实现后台提交请求主要有 3 种方式：remoteLink、formRemote 和

submitToRemote，下面将分别进行介绍。

通过 remoteLink 构造链接，单击该链接，会执行请求，并在请求成功后更新页面内的元素。以添加购物车为例（不能使用 Web Flow 的版本），单击链接，将商品放入购物车，并更新购物车内容。

GSP 页面的链接为：

```
<g:remoteLink  action="addToCart"
    params="${[goodsId: goods.id]}" update="shoppingCart" >
    ${message( code:'cart.addToCart')}
</g:remoteLink>
```

其含义为，当 addToCart action 执行完毕后，用其返回内容（HTML 页面）更新当前页面上 id 为 shoppingCart 的元素（<div id="shoppingCart"> </div>）。因此，还需要修改页面，使购物车内容存放到<div id="shoppingCart">中：

```
<div id="shoppingCart">
    <g:render template="/showCart" model="${[cart:cart]}"/>
</div>
```

相应的 Controller 中的 addToCart action 也要做一些修改，从而将购物车的内容输出到页面上：

```
def addToCart = {
    if(session.userId && params.goodsId) {
        orderService.addToCart(session,params.goodsId)
    }
    if(params.forwardURI)
        redirect(url : params.forwardURI)
    else{
        render (template:"/showCart",
            model:[cart:orderService.prepareCart(session)])
    }
}
```

添加到购物车的逻辑已经改造完成。不需要刷新页面就可以将商品添加到购物车中。而从购物车中移除商品和修改购物车中商品数量的逻辑，也可以进行改造。修改购物车中商品数量，用 submitToRemote 标签进行改造：

```
<g:form controller="goods" action="changeItemNumber
    name="changeItemNumber" method="post" update="shoppingCart">
    <input type="hidden" name="id" value="${lineItem.id}">
    <input type="text" size="2" name="itemNumber"
    value="${lineItem.itemNumber}" >
    <g:submitToRemote name="removeFromCart" update="shoppingCart"
        value="${message (code:'cart.changeItemNumber')}" />
</g:form>
```

将原来的 submit 按钮替换成 submitToRemote 标签，可以实现后台提交表单，并在执行完成后更新指定的页面元素 id（***update="shoppingCart"***）。

从购物车中移除商品的逻辑，不妨用 formRemote 标签进行改造：

```
<g:formRemote url = "${[controller:'goods',action:'removeFromCart']}"
        name="removeFromCart" update="shoppingCart" method="post">
        <input type="hidden" name="id" value="${lineItem.id}">
        <input type="submit" value="${message (code:'cart.remove')}" onclick=
        "return confirm('${message(code:'confirm.removeFromCart')}')"/>
</g:formRemote>
```

将 form 标签替换成 g:formRemote 标签（要注意接收表单的 url 的写法），并指定要更新的元素 id（update="shoppingCart"）即可。

改造完页面，action 的程序也需要进行修改以输出购物车。和 addToCart 的改造原理相同，将原来的 redirect 换成 render 即可：

```
render (template:"/showCart",
    model:[cart:orderService.prepareCart(session)])
```

虽然使用 Grails 提供的标签可以提高 Ajax 的开发效率，但不管怎么说，Server 端的框架，对客户端脚本的封装总是有限的。想完全不写 JavaScript 就开发出具有完美用户体验的 Web 应用是不现实的，JavaScript 就像 HTML CSS 一样，都是进行 Web 开发的必备的基础。

14.4 本章小结

本章首先对 GSP 的知识点进行了总结，然后着重介绍了如何开发自定义的 GSP 标签。本章的最后部分对在 GSP 中开发 Ajax 的常用标签进行了介绍。

第15章 实现 Web Service

Web Service 本质上是要实现远程的程序调用。当前，SOA 的概念非常流行，但从技术层面来讲，并无太多创新之处，本质上都是基于某种通用的远程调用技术。Grails 作为新兴的一栈式快速 Web 开发框架，自然对 REST、SOAP 等技术提供了良好的支持。

15.1 REST 风格的 Web Service

15.1.1 什么是 REST

REST（Representational State Transfer，表述性状态转移）是 Roy Fielding 博士于 2000 年在他的博士论文中提出来的一种软件架构风格。REST 是一种设计风格而不是一个技术标准，它通常基于 HTTP、URI、XML 以及 HTML 这些现有的广泛流行的协议和标准[8] [9]。

REST 从资源的角度来观察整个网络，分布在各处的资源由 URI 确定，而客户端的应用通过 URI 来获取资源的表形。随着不断获取资源的表述，客户端应用不断地在转变着其状态，这就是所谓的表述性状态转移。

REST 定义了应该如何正确地使用（这和大多数人的实际使用方式有很大不同）Web 标准，例如 HTTP 和 URI。如果在设计应用程序时能坚持 REST 原则，那就预示着将会得到一个使用优质 Web 架构的系统。REST 的 5 条关键原则列举如下：

（1）为所有“事物”定义 ID；

（2）将所有事物链接在一起；

（3）使用标准方法；

（4）资源的多重表述；

（5）无状态通信。

15.1.2 在 Grails 中实现 REST

由于 REST 本身并不表示具体的技术，而体现为一种范式或者思想。就以上提到的 5 点，在 Grails 中需要通过技术手段支持的是“使用标准方法”和“资源的多重表述”。

在 Grails 中实现这两点并不复杂。主要基于 Grails 提供的 URL Mapping、XML 与 JSON 输出、XML 格式的表单参数解析、内容协商 4 个基本功能。

“使用标准方法”是通过 URL Mapping 技术实现的。使用不同的方法对资源进行请求时，将调用不同的 action，从而执行不同的具体行为，例如：

```
static mappings = {
  "/goods/$id?"(controller:"product"){
     action = [GET:"show", PUT:"update", DELETE:"delete", POST:"save"]
  }
}
```

此时，当客户端（即调用服务的程序，如浏览器、JavaScript 程序以及其他支持 HTTP 请求的程序）分别使用 GET、PUT、DELETE、POST 方法请求/goods/$id 这一 URI 时，Controller 会相应地执行 show、update、delete 和 save action。

“资源的多重表述”要求能够以多种形式（格式）输出资源，而 Controller 中提供了便捷的以 XML 和 JSON 等格式输出数据的方法。

```
import grails.converters.*
class GoodsController {
    def show = {
        if(params.id && Goods.exists(params.id)) {
            def p = Goods.findByName(params.id)
            render p as XML
        }
        else {
            def all = Goods.list()
            render all as XML
        }
    }
}
```

此时，show action 会以 XML 的形式，返回对应的商品信息。Grails 还可以通过 Controller 的 withFormat 方法，实现根据与 Client 的协商结果返回需要资源的格式。

```
withFormat {
    html bookList:books
    js { render books as JSON }
    xml { render books as XML }
}
```

此时，可以根据 Client 请求的 HTTP 头信息中的 Accept 指定接收类型，也可以根据 URI 的扩展名（如/goods/1.xml），还可以根据 format 参数指定接收类型（如/goods/1?format=xml）。

“资源的多重表述”还有很重要的一个要求，就是支持 XML 格式的数据提交。以 save action 为例，如果 Client 希望保存一条 goods 的记录，将会提交如下的 XML 数据：

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<goods>
    <title>MacBook</ title>
    <description>apple's computer</description>
    <price>10000</price>
    <category id="2"></category>
</goods>
```

相应地，Controller 中接收数据程序代码也是不可思议的简单。原来，Controller 中的 params 可以接收 XML 格式的数据，如：

```
def g = new Goods(params['goods'])
```

15.1.3　在 Client 端调用服务

从本质上讲，REST 方式发布的服务就是基于 HTTP 请求的 Web 资源，Client 可以通过 HTTP 请求访问资源，从而相当于对服务的调用。这里以 Java 开发客户端进行说明。

GET 方式请求资源：

```
URL url = new URL("http://localhost:8080/GDepot/goods");
URLConnection urlConnection = url.openConnection(); //默认就是 GET
urlConnection.setRequestProperty("accept", "text/xml");
InputStream in = urlConnection.getInputStream()));
```

POST 方式保存资源：

```
try {
    String xmlText  = "<goods>"+
        "<title>MacBook</ title>"+
        "<description>apple's computer</description>"+
        "<price>10000</price>"+
        "<category id=\"2\"></category>"+
        "</goods>";

    URL url = new URL("http://localhost:8080/GDepot/goods");
    HttpURLConnection conn = (HttpURLConnection)url.openConnection();
    conn.setDoOutput(true);
    conn.setRequestMethod("POST"); //指定方法为 post
    conn.setRequestProperty("Content-Type", "text/xml");
    OutputStreamWriter wr =
        new OutputStreamWriter(conn.getOutputStream());
    wr.write(xmlText);
    wr.flush();
    wr.close();
```

```
        BufferedReader rd = new BufferedReader(new
            InputStreamReader(conn.getInputStream()));
        String line;
        while ((line = rd.readLine()) != null) {
            System.out.println(line);
        }

        rd.close();
    } catch (Exception e) {
        System.out.println("Error" + e);
    }
```

PUT 方式和 DELETE 方式与 POST 类似，只需要在 conn.setRequestMethod()指定 HTTP 方法为"PUT"、"DELETE"。

15.2 基于 SOAP 的传统 Web Service

SOAP（Simple Object Access Protocol，简单对象访问协议）方式的 Web Service 是 Web Service 传统实现方式，也得到很多大公司的支持。SOAP 是一种标准化的通信规范，主要用于 Web Service 中。SOAP 的出现是为了实现不同应用程序之间能够通过 HTTP 协议，以 XML 格式进行交互，使其与编程语言、平台和硬件无关。此标准由 IBM、Microsoft、UserLand 和 DevelopMentor 在 1998 年共同提出，并得到 IBM、Lotus、Compaq 等公司的支持，然后于 2000 年提交给 W3C。目前的 SOAP 仍然是业界的标准，属于第二代的 XML 协议（第一代有代表性的技术为 XML-RPC 以及 WDDX）。

Grails 内置没有提供对 SOAP 的支持，但可以通过插件很好地支持 SOAP。XFire 插件使用开源的 XFire 框架，其突出的特点就是简单快捷。

使用 XFire 插件[1]，可以将 Grails 中创建的 service 发布成为 Web Service，例如：

```
class GoodsService {
  static expose=['xfire']
  Goods[] getGoodsList(){
    Goods.list() as Goods[]
  }
}
```

此时访问 http://localhost:8080/GDepot/services/goods?wsdl（注意 URL 与 Service 名称的对应关系），可以看到该服务的 WSDL 的内容，如图 15-1 所示。

到此，发布服务的任务已完成，并且可以在 Client 端进行调用了。调用 Web Service 的方法和步骤与普通的 Java 程序并无太大区别，这里不再赘述。

[1] 插件的安装和使用，见下一章。

```
-<wsdl:definitions targetNamespace="http://DefaultNamespace">
 -<wsdl:types>
  -<xsd:schema attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace="http://DefaultNamespace">
   -<xsd:element name="getGoodsList">
      <xsd:complexType/>
    </xsd:element>
   -<xsd:complexType name="ArrayOfGoods">
    -<xsd:sequence>
       <xsd:element maxOccurs="unbounded" minOccurs="0" name="Goods" nillable="true" type="tns:Goods"/>
     </xsd:sequence>
    </xsd:complexType>
   -<xsd:complexType name="Goods">
    -<xsd:sequence>
       <xsd:element minOccurs="0" name="category" nillable="true" type="tns:Category"/>
       <xsd:element minOccurs="0" name="description" nillable="true" type="xsd:string"/>
       <xsd:element minOccurs="0" name="id" nillable="true" type="xsd:long"/>
       <xsd:element minOccurs="0" name="photoUrl" nillable="true" type="xsd:string"/>
       <xsd:element minOccurs="0" name="price" nillable="true" type="xsd:decimal"/>
       <xsd:element minOccurs="0" name="title" nillable="true" type="xsd:string"/>
     </xsd:sequence>
    </xsd:complexType>
   -<xsd:complexType name="Category">
    -<xsd:sequence>
       <xsd:element minOccurs="0" name="categoryName" nillable="true" type="xsd:string"/>
       <xsd:element minOccurs="0" name="goods" nillable="true" type="tns:ArrayOfGoods"/>
       <xsd:element minOccurs="0" name="id" nillable="true" type="xsd:long"/>
     </xsd:sequence>
    </xsd:complexType>
   -<xsd:element name="getGoodsListResponse">
    -<xsd:complexType>
     -<xsd:sequence>
        <xsd:element maxOccurs="1" minOccurs="1" name="out" nillable="true" type="tns:ArrayOfGoods"/>
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
   </xsd:schema>
  </wsdl:types>
  ......
```

图 15-1 使用 firefox 浏览 Web Service

15.3 本章小结

本章介绍了 Web Service 相关的技术，包括 REST 方式以及传统的 SOAP 方式。Grails 还提供了一些插件，可以实现对其他的一些远程调用技术进行整合。例如，xmlrpc、RMI、Hessian、Burlap、Spring's HttpInvoker 等。

第16章 使用 Grails 插件

Grails 是一个一栈式的框架，希望能够解决 Web 开发中多方面的问题。但是技术在不停地革新变化，封闭体系永远不可能做到“包罗万象”。因此，插件扩展成为 Grails 的一个重要特性，它的存在成就了 Grails 几乎无限的扩展可能。正因为如此，读者有必要学习一下 Grails 的插件机制，了解常用的插件，并掌握一定的插件开发技能。

由于开发 Grails 插件需要有高级的 Groovy 知识做基础，本章暂不介绍如何开发 Grails 插件。相关的知识将在本书的最后部分进行介绍。

16.1 插件的安装

Grails 提供了大量插件，涉及安全、UI、Web Service、性能、测试、调试等多个方面。通过官方网站可以看到插件列表[10]，如图 16-1 所示。

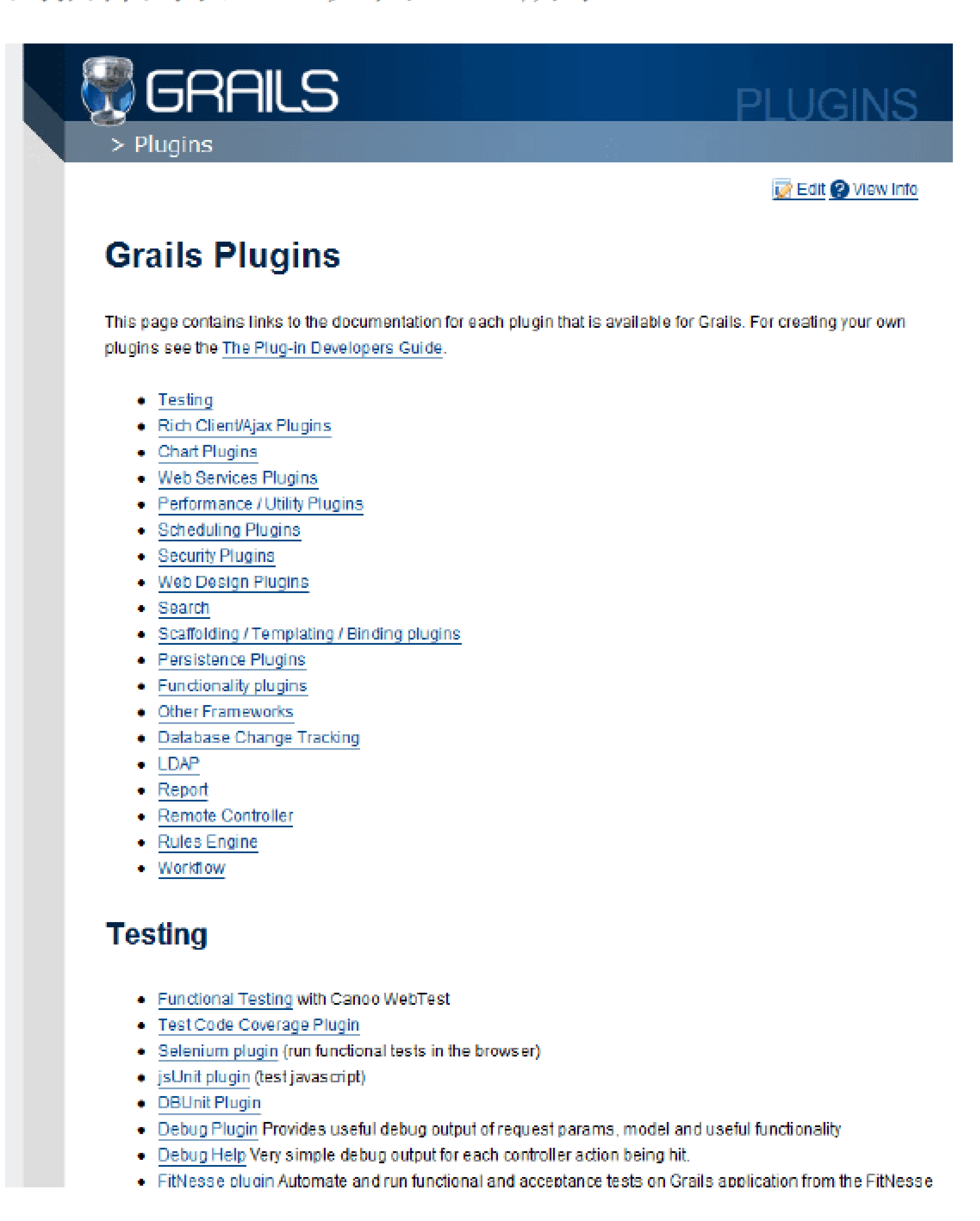


图 16-1　官方网站列出插件的页面

使用 Grails 的命令行也可以查看在线的插件资源，list-plugins 命令可用于查看插件列表[1]。list-plugins 命令在读取插件信息后会显示：插件名<当前最新版本号>--简单插件描述。

```
>grails list-plugins
Welcome to Grails 1.0.4 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: D:\grails-1.0.4
Base Directory: D:\workspace\GDepot
Note: No plugin scripts found
Running script D:\grails-1.0.4\scripts\ListPlugins.groovy
Environment set to development

Plug-ins available in the Grails repository are listed below:
-------------------------------------------------------------

acegi               <0.4.1>         --  Grails Spring Security 2.0 Plugin
activemq            <0.0>           --  Grails Activemq Plugin
aop                 <no releases>   --  No description available
audit-logging       <0.4>           --  adds hibernate audit logging and
                                        onChange event handlers to GORM
                                        domain classes
authentication      <1.0>           --  Simple, extensible authentication
                                        services with signup support
auto-delegator      <0.1>           --  This plugin automatically does
                                        delegation
autobase            <0.6>           --  Automate your database work as much
                                        as possible
autorest            <no releases>   --  No description available
avatar              <0.3>           --  Grails Avatar Plugin
axis2               <0.5>           --  Adds Web Service support for Grails
                                        services using Apaxhe Axis2.
background-thread   <1.3>           --  Background Thread Plugin
bubbling            <1.5.0>         --  Bubbling Library YUI Extension
cache               <no releases>   --  No description available
cas-client          <1.0>           --  This plugin provides client
                                        integration for JA-SIG CAS
clamav              <0.1>           --
code-coverage       <0.9>           --  Generates Code Coverage reports
compress            <0.2>           --  Servlet filter to Compress content.
converters          <0.3>           --  Provides JSON and XML Conversion for
                                        common Objects (Domain Classes,
                                        Lists, Maps, POJO)
criteria            <1.0>           --  Criteria (record filtering) plugin
```

[1] 执行 list-plugins 命令要求计算机能够访问 internet。

```
crowd                 <0.4>          -- Plugin that adds support for
                                        Atlassian Crowd authentication and
                                        authorization to a Grails webapp.
datasources           <0.2>          -- Grails Datasources Plugin.
dbmapper              <0.1.7>        -- Database relation to domain object
                                        generator
dbmigrate             <0.1.5>        -- Database migration tasks
dbunit-operator       <0.2>          -- DBUnit Operator
debug                 <1.0.2>        -- Provides debug tools for
                                        development mode
debug-help            <0.1>          -- Debugging Help
djangotemplates       <no releases>  -- No description available
dojo                  <0.4.3>        -- Provides integration with the Dojo
                                        toolkit http://dojotoolkit.org, an
                                        Ajax framework.
domain-schema         <no releases>  -- No description available
drilldowns            <1.0>          -- Drilldown(summary to detail) plugin
drools                <0.1>          -- This plugin provides Drools Rule
                                        Based Management System function-
                                        ality to Grails application.
dwr                   <0.1>          -- This plugin adds DWR capabilies to
                                        the services in a Grails
                                        application.
eastwood-chart        <0.3>          -- This plugin wraps JFreeChart's
                                        Eastwood servlet for Grails app
easybtest             <0.2>          -- Plugin summary/headline
echo2                 <0.1>          -- Echo2 capabilities to Grails
                                        applications
exchange-rates        <1.0>          -- Foreign currency exchange rates
                                        using Yahoo!
ext                   <2.0.2>        -- Ext JavaScript Library
ext-ui                <no releases>  -- No description available
extended-data-binding<0.2>           -- This plugin extends Grails' data
                                        binding possibilities, both for
                                        parsing the user input and to present
                                        formatted data
fckeditor             <0.8>          -- Fckeditor
feeds                 <1.4>          -- Render RSS/Atom feeds with a simple
                                        builder
filter                <0.2>          -- Dynamic filter plugin
fixtures              <0.4>          -- fixtures
flash-player          <1.0>          -- Grails TagLib plugin for the JW FLV
                                        Media Player
flex                  <0.2>          -- Provides integration between Grails
                                        and Flex
```

```
func              <1.7>                 -- Plugin to make your application
                                           ables to call Func (https://
                                           fedorahosted.org/func)
google-chart      <0.4.8 (?)>           -- This plugin adds Google Chart API
                                           features to Grails applications.
grails-ui         <1.0.2-SNAPSHOT>      -- Grails UI
grid-gain         <0.1.1>               -- GridGain Plugin to enable GridGain
                                           2.1 (snapshot) support
gwt               <0.2.4>               -- The Google Web Toolkit for Grails.
help-balloons     <1.1>                 -- Grails TagLib plugin for the
                                           HelpBalloon system from Beau Scott
honey             <no releases>         -- No description available
i18n-gettext      <0.4>                 -- I18n gettext plugin for grails
i18n-templates    <1.0.4>               -- I18n Templates
include           <0.3>                 -- This plugin adds include behaviour
                                           to Grails application.
iui               <0.2>                 -- iUI for Grails
ivy               <2.0.0-rc1>           -- The Ivy Dependency Manager for
                                           Grails.
j2d               <0.2>                 -- Enables Java2D rendering using
                                           GraphicsBuilder
jackrabbit        <no releases>         -- No description available
jasper            <0.9>                 -- Easily launch Jasper reports from
                                           a Grails application.
jawr              <2.51>                -- adds Jawr (https://jawr.dev.java.
                                           net) functionality to Grails
                                           applications.
jbpm              <0.1>                 -- Jbpm plugin
jcaptcha          <1.0>                 -- Grails JCaptcha Plugin
jcr               <0.2-SNAPSHOT (?)>    -- This plugin provides JCR-based
                                           persistence for Grails
jms               <0.3>                 -- This plugin adds MDB functionality
                                           to services.
jmx               <0.4>                 -- The JMX Grails Plugin
jquery            <1.0RC1>              -- JQuery for Grails
jsecurity         <0.3>                 -- Security support via the JSecurity
                                           framework.
jsf               <no releases>         -- No description available
jsunit            <0.3>                 --
jttaglib          <0.2>                 --
laszlo            <0.6.5>               -- OpenLaszlo plugin for Grails
ldap              <0.8>                 -- Adds easy to use LDAP connectivity
license           <0.1>                 -- This plugin provides license
                                           management to Grails application.
liquibase         <1.8.1.0>             -- LiquiBase Database Refactoring for
```

```
                                    Grails
localizations   <1.0>       --  Localizations (messages) plugin
logging         <0.1>       --  Grails Dynamic Logging Plugin
lookups         <1.0>       --  Application lookup (reference
                                tables) plugin
mail            <0.5>       --  Provides Mail support to a running
                                Grails application
menus           <1.1>       --  Menus plugin
modalbox        <0.3>       --  This plugin adds the ModalBox to your
                                Grails applications.
mondrian        <0.1>       --  This plugin installs Pentaho Mondrian
                                into your Grails application.
mootools        <0.1>       --  Provides integration with the Mootools
                                toolkit http://mootools.net, a light OO
                                javascript framework.
oauth           <0.1.1>         --  Adds OAuth capability to Grails apps
ofchart         <0.5>           --
ohloh           <no releases>   --  No description available
openid          <0.1>           --  OpenID
p6spy           <0.4>           --  This plugin adds p6spy to your
                                    application
paypal          <0.2>           --  A Grails plug-in that provides
                                    integration with Paypal's Instant
                                    Payment Notfication (IPN) system
portlets        <no releases>   --  No description available
post-code       <1.0>           --  Provides a service for getting lat/lon
                                    from UK postcodes and calculating
                                    distance between two postcode centres
profiler        <0.1>           --  Profiles a Grails application on demand.
promo           <0.1>           --
quartz          <0.4.1-SNAPSHOT>--  This plugin adds Quartz job scheduling
                                    features to Grails application.
radeox          <0.1>           --  A plugin for the Radeox Wiki-render
                                    engine to allow easy markup and
                                    cross-linking.
recaptcha       <0.3.1>         --  This plugin adds ReCaptcha support to
                                    Grails.
ref-code        <0.1>           --
remoting        <1.0>           --  Adds easy-to-use server-side and
                                    client-side RPC support.
restore-state   <1.0>           --  Restore State Plugin
richui          <0.5>           --  Provides a set of AJAX components
runtime-logging <0.3>           --  Grails Runtime Logging Plugin
scaffold-tags   <0.7>           --  Adds tags to support fully-customizable
                                    & dynamic scaffolding
```

```
searchable          <0.5.1>        -- Adds rich search functionality to
                                      Grails domain models. This version
                                      is recommended for JDK 1.5+
searchable14        <0.5.1>        -- Adds rich search functionality to
                                      Grails domain models. This version
                                      is recommended for JDK 1.4
selenium            <0.5>          --
settings            <1.0>          -- Application settings (global
                                      constants) plugin
shopping-cart       <0.2>          -- Shopping cart plugin
skinnable           <0.2>          -- This plugin adds skinnable support
                                      to Grails application.
smartclient         <no releases>  -- No description available
sound-manager       <0.2>          -- Grails SoundManager Plugin
sparkline           <0.1>          -- Plugin summary/headline
springcache         <1.0>          -- Spring Cache Plugin
springmvc           <0.1>          -- Grails Spring MVC Controller
                                      Plugin.
stark-security      <0.3.1>        -- A simple yet robust implementation
                                      of Spring Security for Grails
static-resources    <0.5>          --
struts1             <1.3.8.1>      -- Provides integration between Grails
                                      and the Struts 1 framework
syntax-highlighter  <0.1.2>        -- Syntax Highlighter Plugin
terracotta          <0.1>          -- Simple Tarracotta Integration
test-template       <1.3.2>        -- Test Templates
testing             <0.4>          -- Unit testing plugin
triggers            <no releases>  -- No description available
ui-performance      <1.0>          -- Grails UI Performance Plugin
webtest             <0.5.1>        -- A plug-in that provides functional
                                      testing for Grails using Canoo Web
                                      Test
webxml              <1.2>          -- Create useful additions to web.xml
wicket              <0.6>          -- Provides integration between Grails
                                      and the Wicket framework
xfire               <0.7.5>        -- Add Web Service support for Grails
                                      services using XFire.
xmlrpc              <0.1>          -- This plugin adds XML RPC
                                      functionality to GRAILS
xtemplates          <0.1>          -- Extensible (eXtreme) Templates
yui                 <2.6.0>        -- Yahoo! User Interface Library (YUI)

To find more info about plugin type 'grails plugin-info [NAME]'
To install type 'grails install-plugin [NAME] [VERSION]'
For further info visit http://grails.org/Plugins
```

犹豫再三，选择了保留全部 list-plugins 命令执行时的输出结果。因为没有什么能比这

个更能反映当前 Grails 插件的繁荣程度了。当然在面对大量的插件时，没有必要感到无所适从，毕竟插件存在的目的就是帮助用户解决问题或者简化问题。查看插件列表后，可以使用 plugin-info 命令查看插件的详情。如：

```
>grails plugin-info acegi
```

在确定要在项目中使用某一插件后，则需要安装插件，可以使用 install-plugin 命令安装插件。install-plugin 命令非常强大，能够完成对插件的下载和安装，也支持安装本地的插件程序。使用格式如下：

```
>grails install-plugin [URL 或 本地文本名]
```

或者

```
>grails install-plugin 插件名 版本号
```

如果不指定版本号，则安装当前最新的稳定版本。

16.2 插件的组织结构

每一个 Grails 的插件更像是一个“小”的 Grails 应用。它的组织结构如图 16-2 所示。

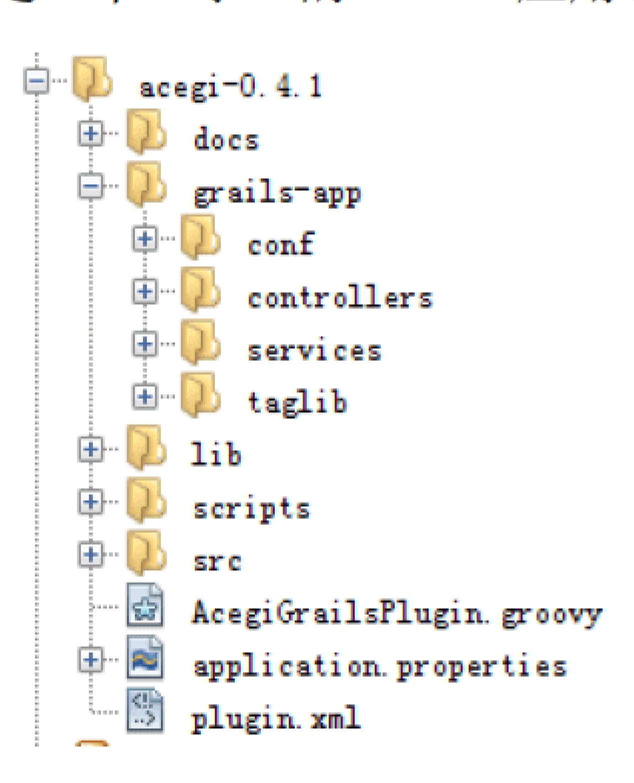


图 16-2 Grails 插件的组织结构

可以看出，每个插件都可以有它自己的配置文件、控制器、Service、标签库、自定义命令（Script）以及测试程序等。但一般来说，Grails 插件中并不包含下面的内容：

```
grails-app/conf/DataSource.groovy
grails-app/conf/UrlMappings.groovy
web-app/WEB-INF/*
```

对于 UrlMappings.groovy，如果读者希望在插件中自己定义 Url 映射，可以定义为 XXXUrlMappings.groovy，例如 BlogUrlMappings.groovy。而对于希望安装到 web-app\WEB-INF\目录中的内容，推荐的做法是修改 scripts_Install.groovy（Gant 脚本），将它们安

装到全局应用程序的 web-app\WEB-INF\中。

接下来，介绍几个常见插件的使用。

16.3 插件的使用

16.3.1 Acegi 插件

Acegi 是 Spring 项目的一个子项目，更新到 2.0 时这个项目被更名为 Spring Security。Grails 的 Acegi 插件，虽然名称仍为 Acegi，但本质上是对 Spring Security2.0 进行了封装。使用 Acegi 插件，可以方便地解决权限分配、权限验证等一系列与安全相关的问题。

首先，需要安装 Acegi 插件：

```
>grails install-plugin acegi
```

Grails 会完成下载，并执行解压、编译、安装等一系列动作。如果下载速度太慢，不妨使用其他下载工具下载，下载地址为：http://plugins.grails.org/grails-acegi/tags/RELEASE_0_4_1/grails-acegi-0.4.1.zip。下载完成后使用命令 grails install-plugin D:\grails-acegi-0.4.1.zip 进行安装（假设该插件文件存放在 D 盘根目录）。

Spring Security 解决的是用户、角色、资源之间的关系。用户与角色是多对多的关系，角色与资源（权限）也是多对多的关系。它们之间的关系可以通过多种方式进行存储，例如数据库、静态配置文件等。

其次，使用插件提供的命令来创建用户、角色以及资源的 Domain。命令格式如下：

```
>grails create-auth-domains AuthUser Role
```

其中的 AuthUser 是用于注册和登录的“用户”Domain，而 Role 则对应为角色类。命令成功执行后，会创建几个 Groovy 文件，分别是 AuthUser.groovy、Role.groovy、Requestmap.groovy 三个 Domain 类和 SecurityConfig.groovy 配置文件，以及 login 和 logout 两个 controller。

用户类的内容如下：

```
class AuthUser {
    static transients = ['pass']
    static hasMany = [authorities: Role]
    static belongsTo = Role

    /** Username */
    String username
    /** User Real Name */
    String userRealName
    /** MD5 Password */
```

```
    String passwd
    /** enabled */
    boolean enabled

    String email
    boolean emailShow

    /** description */
    String description = ''

    /** plain password to create a MD5 password */
    String pass = '[secret]'

    static constraints = {
        username(blank: false, unique: true)
        userRealName(blank: false)
        passwd(blank: false)
        enabled()
    }
}
```

角色类的内容如下：

```
class Role {

    static hasMany = [people: AuthUser]

    /** description */
    String description
    /** ROLE String */
    String authority = 'ROLE_'

    static constraints = {
        authority(blank: false)
        description()
    }
}
```

从两个类的内容也可以看出它们之间存在多对多的关系。

Requestmap 类描述了资源与角色的关系，但是它没有使用数据库中的多对多关系，因为那样显得有些过于复杂。事实上，Requestmap 只是记录了 url 和用逗号分隔的多个角色。Requestmap 的内容如下：

```
class Requestmap {
    String url
    String configAttribute /* 记录用逗号分隔的多个角色 */
```

```
        static constraints = {
            url(blank: false, unique: true)
            configAttribute(blank: false)
        }
    }
```

虽然有了上面的 Domain 类，用户已经可以自己实现按角色分配资源，并且将分配方式记录在数据库中，但是 Acegi 插件提供了更简便的方法，使用 generate-manager 命令可以自动生成维护用户-角色-资源的页面：

```
>grails generate-manager
```

命令执行成功，会自动生成如下内容：

```
grails-app/controllers/RequestmapController.groovy
grails-app/controllers/RoleController.groovy
grails-app/controllers/UserController.groovy
grails-app/views/requestmap/create.gsp, edit.gsp, list.gsp, view.gsp
grails-app/views/role/create.gsp, edit.gsp, list.gsp, view.gsp
grails-app/views/user/create.gsp, edit.gsp, list.gsp, view.gsp
```

此时，可以分别使用 User、Role、Requestmap 的 CRUD 页面进行查看、添加、修改和删除。首先访问 http://localhost:8080/GDepot/role/create，并创建两个 Role：user 和 admin，如图 16-3、图 16-4 所示。



图 16-3　创建 user 角色



图 16-4　创建 admin 角色

然后，使用 http://localhost:8080/GDepot/requestmap/create 配置限制访问的资源，如图 16-5 所示。

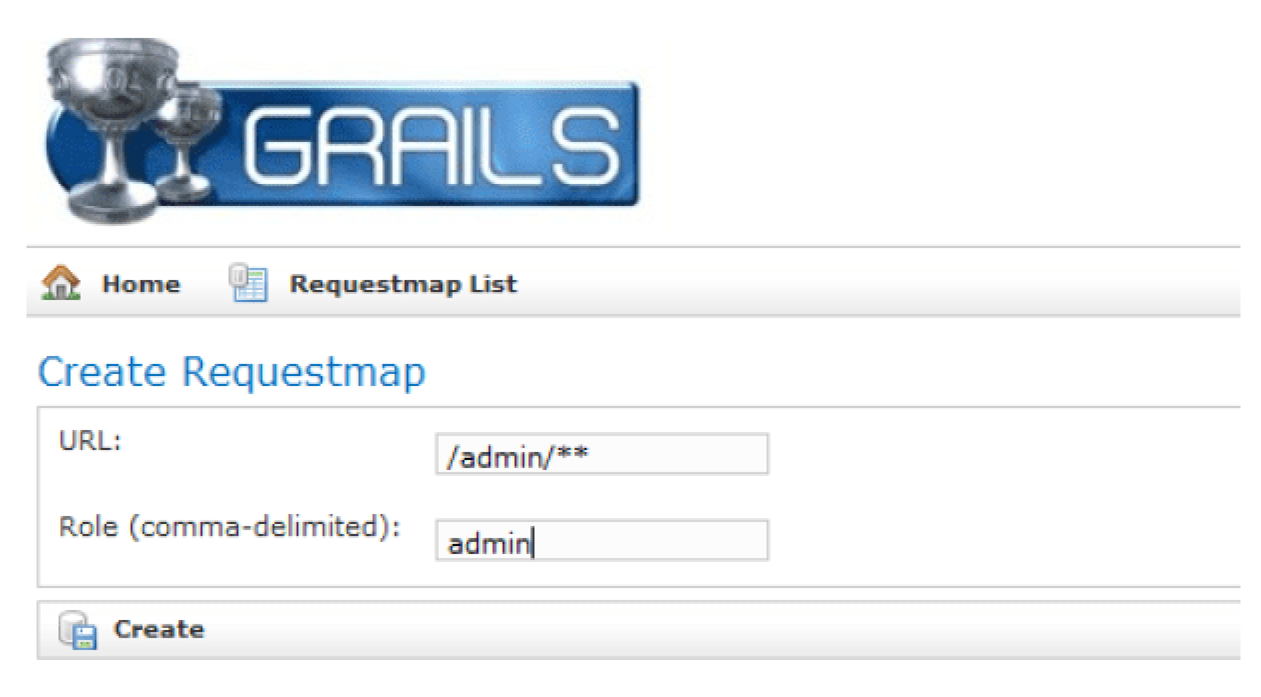


图 16-5　配置 Requestmap

此时，任何/admin/下的资源，都需要有 admin 角色才能访问。接下来，使用 http://localhost:8080/GDepot/user/create 页面创建一个用户，只为它配置 user 角色，如图 16-6 所示。

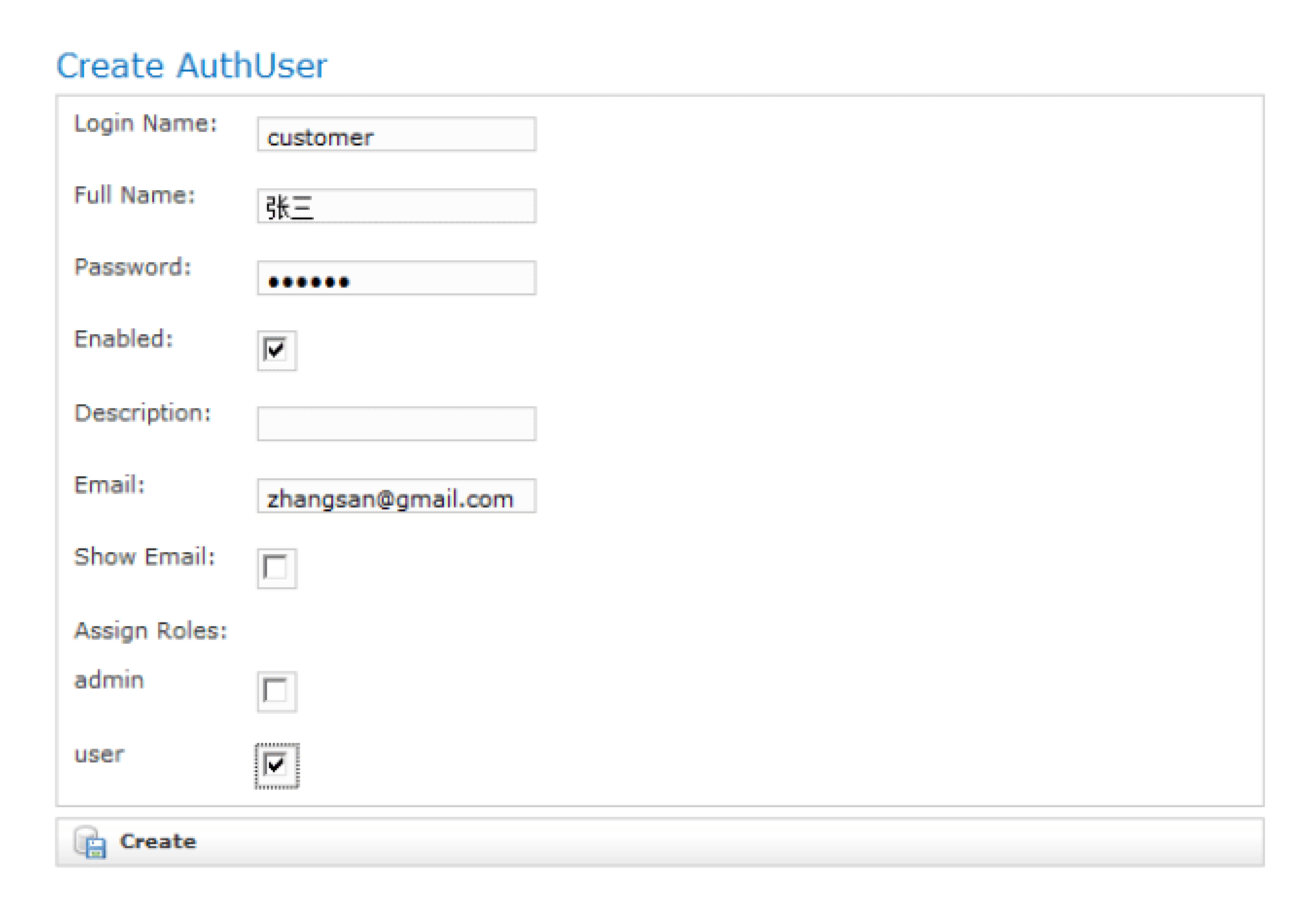


图 16-6　创建 user 角色的用户

然后使用该用户登录 http://localhost:8080/GDepot/login/auth，登录成功后，可以发现，此时并不能访问/admin/下面的资源，如图 16-7 所示。

HTTP ERROR: 403

```
Access is denied
```

RequestURI=/GDepot/admin/listGoods

Powered by Jetty://

图 16-7　访问 http://localhost:8080/admin/listOrders 时报错

而当使用拥有 admin 角色的用户登录时，就可以访问该页面了。

有的时候并不需要动态地配置资源和角色对应关系，此时可以在 SecurityConfig.groovy 中静态地进行配置：

```
requestMapString = """
    CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
    PATTERN_TYPE_APACHE_ANT

    /login/**=IS_AUTHENTICATED_ANONYMOUSLY
    /admin/**=ROLE_USER
    /book/test/**=IS_AUTHENTICATED_FULLY
    /book/**=ROLE_SUPERVISOR
    /**=IS_AUTHENTICATED_ANONYMOUSLY
"""
```

在 UI 上创建 user 角色，它存放到数据库的名称就是 ROLE_USER；同理，admin 角色在数据库中的名称就是 ROLE_ADMIN。

回顾图 16-6 创建用户，显然那个页面是不能给普通用户使用的，因为不可能允许普通用户自行配置角色。这里需要为普通用户单独开发一个用于注册的页面。acegi 插件的 generate-registration 命令可以帮助完成这个任务：

```
>grails generate-registration
```

运行成功后，会自动生成如下文件：

```
grails-app/controllers/CaptchaController.groovy
grails-app/controllers/RegisterController.groovy
grails-app/services/EmailerService.groovy
grails-app/views/register/edit.gsp, index.gsp, show.gsp
```

这个自动生成的注册页面实际上是非常完善的，它包含两次输出密码、验证码等，访问 http://localhost:8080/GDepot/register，可看到如图 16-8 所示的注册页面。



图 16-8　自动生成的注册页面

怎么样？是不是后悔在第 6 章中自己手动开发用户注册的功能了？从生成的 Controller 可以看到，使用注册页面注册的用户会被标记为默认角色：

```
def person = new AuthUser()
person.properties = params
def config = authenticateService.securityConfig
def defaultRole = config.security.defaultRole

def role = Role.findByAuthority(defaultRole)
if (!role) {
    person.passwd = ''
    flash.message = 'Default Role not found.'
    render(view: 'index', model: [person: person])
    return
}
```

而默认角色是在 SecurityConfig.groovy 中定义的：

```
defaultRole = 'ROLE_USER'
```

通过 Acegi 插件提供的命令，可以完成用户登录、角色创建、角色分配、资源配置等功能。但与权限相关的问题总是很复杂的，上面的功能还不能解决全部的问题。有的时候，可能需要在同一个页面上对不同角色的用户显示不同的内容。Acegi 插件提供了几个标签，用于识别角色、登录等。这几个标签的介绍如表 16-1 所示。

表 16-1　Acegi 标签库

标签	简介
ifAllGranted ifAnyGranted ifNotGranted	判断当前用户是否拥有角色（角色是以“,”分隔的列表）： ifAllGranted 判断当前用户是否拥有全部角色： <g:ifAllGranted role="ROLE_USER,ROLE_ADMIN"> 这是只有 ROLE_USER 或 ROLE_ADMIN 才能看到的文本。 </g: ifAllGranted> ifAnyGranted 判断当前用户是否拥有某一角色： <g: ifAnyGranted role="ROLE_USER,ROLE_ADMIN"> 这是 ROLE_USER 或 ROLE_ADMIN 都能看到的文本。 </g: ifAnyGranted> ifNotGranted 判断当前用户是不是不能拥有某一角色： <g: ifNotGranted role="ROLE_USER,ROLE_ADMIN"> 这是除了 ROLE_USER 和 ROLE_ADMIN 都能看到的文本。 </g: ifNotGranted>

续表

标签	简介
isLoggedIn isNotLoggedIn	判断是否登录 <g:isLoggedIn> 您是已登录的用户 </g: isLoggedIn> <g:isLoggedIn> 您还没有登录 </g: isLoggedIn>

除了上面的标签，Acegi 插件还提供了一个 Service 类（AuthenticateService），该类包含了一些非常实用的方法：

```
principal() 可以获取当前登录用户的主要信息(包含 username, authorities)
userDomain() 可以获取当前登录用户的 Domain 类
getSecurityConfig() 可以获取安全配置信息
passwordEncoder(String passwd) 对密码进行加密编码
```

一般来说，设计真正安全的系统，就需要系统各层都是安全的。因此，还需要保证业务逻辑层同样安全。Acegi 插件还提供了为 Service 类方法定义访问角色的功能，其使用方式是在需要安全检查的方法前添加 Annotation：

```
@Secured(["ROLE_ROLENAME1","ROLE_ROLENAME2"])
```

例如，配置 admin 可以访问 methodA，而配置 user 可以访问 methodB：

```
import org.springframework.security.annotation.Secured;
class SomeService {
  static transactional = true
  static scope = "request"
  @Secured(["ROLE_ADMIN"])
  def methodA(){
    println "methodA() method for ROLE_AMDIN"
    return "this method is for ROLE_ADMIN Only"
  }

  @Secured(["ROLE USER"])
  def methodB(){
    println "methodB() method for ROLE_USER"
    return "this method is for ROLE_USER Only"
  }
}
```

有了上面的基础，就可以实现动态控制（或者静态配置）用户、角色、资源的关系；

也可以精细地控制某一个细小的功能点属于哪个角色。这就已经可以解决绝大多数常见的与权限控制相关的需求了。但 Spring Security 还有更多的功能，它还支持除数据库登录认证外的多种认证方式：LDAP、OpenId 等。感兴趣的读者可以参考 http://static.springframework.org/spring-security/site/index.html。

16.3.2 Debug 插件

正如前面所述，不同的插件解决不同领域的问题。Acegi 插件可以解决与安全相关的问题。而 Debug 插件可以简化调试的任务。Debug 插件是一个很小的插件，但功能却非常实用。安装 Debug 插件的命令为：

```
>grails install-plugin debug
```

Debug 插件默认支持两种方式输出调试信息：在控制台输出和在页面输出。如果希望在控制台输出调试信息，需要先开启 log4j 中相应的配置。首先，在项目的 Config.groovy 文件中添加：

```
logger.grails.app.service.DebugService="info"
```

然后在 Config.groovy 文件中配置希望 Debug 插件记录的调试信息。目前，Debug 插件支持下面 8 种调试信息：

```
grails.debug.system=true
grails.debug.stats=true
grails.debug.params=true
grails.debug.headers=true
grails.debug.controller=true
grails.debug.session=true
grails.debug.requestAttributes=true
grails.debug.model=true
```

可以根据需要，自行将上面的一种或几种调试信息添加到 Config.groovy 中，此时，控制台上会输出相应的内容。

Debug 插件也提供在页面上输出调试信息的办法，只需要简单地写一个标签：

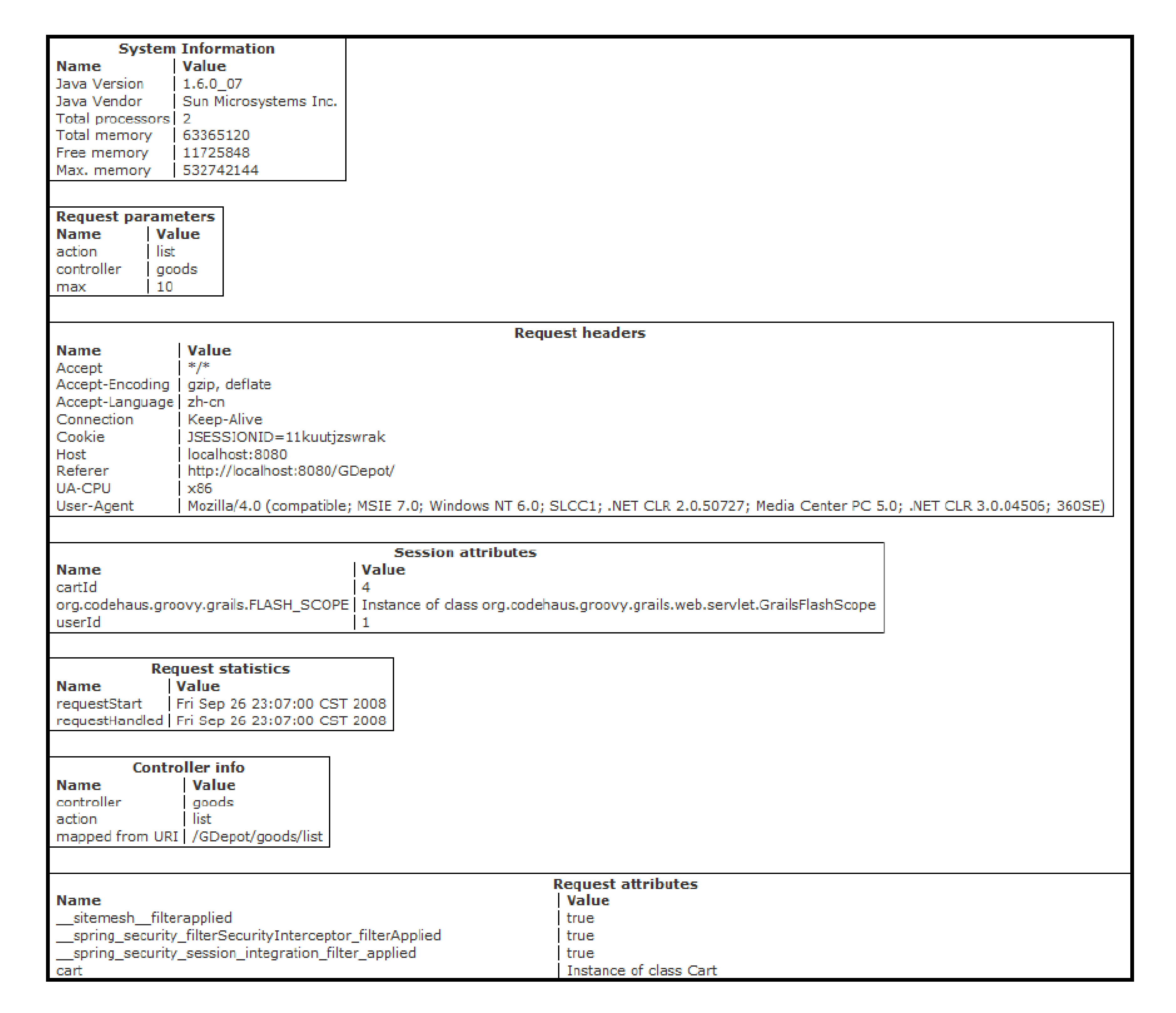
```
<debug:info/>
```

输出效果如图 16-9 所示。

如果希望在项目中禁用 Debug 插件，需要在 Config.groovy 中添加配置项：

```
grails.debug.enabled='false'
```

由于 Debug 插件会输出大量的调试信息，如果在生产环境中使用，会带来严重的性能下降。插件的作者考虑到这一点，因而 Debug 插件默认不会在生产模式下运行。如果需要在生产模式下运行，需要在 Config.groovy 中添加配置项：



System Information

Name	Value
Java Version	1.6.0_07
Java Vendor	Sun Microsystems Inc.
Total processors	2
Total memory	63365120
Free memory	11725848
Max. memory	532742144

Request parameters

Name	Value
action	list
controller	goods
max	10

Request headers

Name	Value
Accept	*/*
Accept-Encoding	gzip, deflate
Accept-Language	zh-cn
Connection	Keep-Alive
Cookie	JSESSIONID=11kuutjzswrak
Host	localhost:8080
Referer	http://localhost:8080/GDepot/
UA-CPU	x86
User-Agent	Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0; SLCC1; .NET CLR 2.0.50727; Media Center PC 5.0; .NET CLR 3.0.04506; 360SE)

Session attributes

Name	Value
cartId	4
org.codehaus.groovy.grails.FLASH_SCOPE	Instance of class org.codehaus.groovy.grails.web.servlet.GrailsFlashScope
userId	1

Request statistics

Name	Value
requestStart	Fri Sep 26 23:07:00 CST 2008
requestHandled	Fri Sep 26 23:07:00 CST 2008

Controller info

Name	Value
controller	goods
action	list
mapped from URI	/GDepot/goods/list

Request attributes

Name	Value
__sitemesh__filterapplied	true
__spring_security_filterSecurityInterceptor_filterApplied	true
__spring_security_session_integration_filter_applied	true
cart	Instance of class Cart

图 16-9　<debug:info/>标签在页面上输出的调试信息

```
grails.debug.productionOverride=anything
```

16.4　本章小结

本章对 Grails 的插件功能进行了简单的介绍，包括插件的安装以及插件的组织结构。最后以 Acegi 插件和 Debug 插件为例，展示了通过插件系统简化项目开发和调试。其中 Acegi 插件是一个非常强大的权限控制管理系统，通过它可以方便地实现对用户、角色、资源进行管理，从而解决与系统安全相关的问题。对比第 6 章自己实现的注册、登录页面，使用 Acegi 的优势就可以充分显现了。Grails 目前已经拥有大量的可用插件，感兴趣的读者不妨多花些时间阅读一下官方的插件列表，相信一定能从中选出一些对自己有帮助的插件。

第四篇

Grails 解密

学习完第二部分，读者应该可以使用 Grails 开发简单的 Web 应用了；学习了第三部分，读者应该掌握了不少 Grails 的高级技能，并且能够进行调优等较高级的操作了，通过使用插件，还可以更轻松地开发更为复杂的功能。

越是这种复杂的功能，越显示出了 Grails 的神奇。但是，在计算机中，并不存在任何神奇的东西、任何奇妙的程序，都是通过一行一行的代码实现的。Grails 的神奇，源于 Groovy 的神奇。本书的第四部分，就打算从 Groovy 的高级特性入手，结合少量的 Grails 源程序，解密为什么 Grails 可以如此的神奇。

第 17 章 高级 Groovy 特性

Grails 的神奇源于 Groovy 的强大。本书一开始就介绍过，Groovy 是一种动态脚本语言，本章将对 Groovy 的动态特性进行阐述。

17.1　动态方法调用与属性访问

17.1.1　动态方法调用

在 Groovy 中，可以使用字符串作为方法名，实现方法调用：

```
class Foo {
    def bar() {
        println "foo bar"
    }
}

def strMethodName = "bar"
def foo = new Foo()
foo."${strMethodName}"()
```

从上例的黑斜体部分可以看到，因为使用了在编译时无法确定其内容的字符串作为方法名去调用方法，所以说此时的方法调用是动态的。

对于包含参数的方法，也可以使用相似的方法进行调用：

```
foo."${strMethodName}"(param1,param2)
```

对于多个参数的情况，可以使用“*”运算符将数组（List）展开：

```
def params = [param1, param2]
foo."${strMethodName}"( *params )
```

17.1.2　动态属性访问

对于 Groovy 对象的属性访问，属性名也可以是在编译时未知的：

```
Class Foo {
    def prop = "value"
}
def strPropertyName = "prop"
def foo = new Foo()
println foo[strPropertyName]
```

因此，通过“[ ]”运算符，可以实现用“[ ]”运算符访问 Groovy 对象的属性。

17.2 invokeMethod 和 getProperty

在 Groovy 中，可以通过重载 invokeMethod、setProperty 和 getProperty 方法，在 Groovy 对象上访问未定义过的方法和属性。

```
class Foo  {
   def bar(){ println "foo bar" }
   def invokeMethod(String name,  args ){
      println "invokeMethod: ${name}"
   }
}
def foo = new Foo()
foo.bar()
foo.hello()
foo.speak()
```

输出内容为：

```
foo bar
invokeMethod: hello
invokeMethod: speak
```

这里并没有在 Foo 类中定义过 hello 和 speak 方法，但执行 foo.hello()和 foo.speak()并不会报错，其奥妙就在于 invokeMethod 方法。invokeMethod 方法会拦截对未知方法的调用。invokeMethod 的第一个参数是被拦截的方法名称，第二个参数是被拦截方法的参数（Object[]类型）。

通过 invokeMethod 方法的支持，可以调用未定义的方法，从而写出很有创意的程序，例如 XML Builder：

```
class XmlBuilder {
  def out
  XmlBuilder(out) { this.out = out }
  def invokeMethod(String name, args) {
     out << "<$name>"
     if(args[0] instanceof Closure) {
```

```
            args[0].delegate = this // L1
            args[0].call()
        }
        else {
           out << args[0].toString()
        }
        out << "</$name>"
    }
}

def xml = new XmlBuilder(new StringBuffer())
xml.html {
    head { //L2
        title "Hello World"
    }
    body {
        p "Welcome!"
    }
}
```

L1 是一个关键点，通过将当前对象 this 设定为闭包对象 delegate，从而使得在闭包中可以调用 this 的方法。L2 在调用 head 方法时，实际上在调用 this 的 head 方法，又由于 XmlBuilder 中不存在 head 方法，invokeMethod 方法会被再次调用。相信读者在真正理解了上面的代码之后，对 Grails 中众多 Builder（如 HibernateCriteriaBuilder、JSON Builder、XML Builder、Spring Builder 等）的实现原理，应该已经有了一定的想法了。

事实上 invokeMethod 也可以拦截“已知”方法（类中定义过的方法）。但要求 Groovy 类必须实现 GroovyInterceptable 接口：

```
class Foo implements GroovyInterceptable {
    def bar(){ println "foo bar" }
    def invokeMethod(String name,  args ){
        println "invokeMethod: ${name}"
        super.invokeMethod(name,args)
    }
}
```

不难想象，在 Groovy 中，可以比较轻松地通过 invokeMethod 方法， 实现 AOP[1]编程。不过 AOP 不是本书的重点，感兴趣的读者可以阅读相关文章[11][12]。

与 invokeMethod 的功能类似，重载 getProperty/setProperty 方法可以访问未定义的属性：

---

[1]AOP（Aspect Oriented Programming，面向切面编程）的目标是通过预编译方式和运行期动态代理实现在不修改源代码的情况下给程序动态统一添加功能。AOP 实际是 GoF 设计模式的延续，设计模式孜孜不倦追求的是调用者和被调用者之间的解耦，AOP 可以说也是这种目标的一种实现

```
class Expandable {
    def storage = [:] // 使用 Map 存放<属性名,属性值>
    def getProperty(String name) { storage[name] }
    void setProperty(String name, value) { storage[name] = value }
}

def e = new Expandable()
e.foo = "bar"
println e.foo
```

methodMissing 和 propertyMissing 方法，与 invokeMethod 和 getProperty/setProperty 方法非常类似，它们的含义是拦截因为方法或者属性不存在而引发的错误。对于 static 的方法和属性，可以通过在 Groovy 对象的 metaClass 上使用 methodMissing 和 propertyMissing 实现动态添加。

17.3 MOP 动态基础

Groovy 的 Team Leader——Guillaume Laforge 说过，MOP (Meta Object Protocol)是他最喜欢的 Groovy 特性。

虽然通过 invokeMethod 可以实现调用未知的属性和方法，但是通过 invokeMethod 并没有真正地为 Groovy 类添加方法[1]。对于动态脚本语言而言，在运行时能够动态地添加方法和属性，是非常普遍的特性。在 Groovy 中，这可以通过 MOP 实现。

17.3.1 遍历方法和属性

MOP 是 Groovy 中的一个比较重要的概念，它是实现 Groovy 动态特性的基础。每个 Groovy 类都包含一个 metaClass 属性，通过 metaClass 可以实现遍历方法和属性，也可以实现动态添加方法和属性。空谈概念，晦涩又无趣，通过下面的简单例程，对 metaClass 能产生一个初步的认识：

```
class Foo{
    def prop
    def bar(){ }
}
def foo = new Foo()
foo.metaClass.methods.each { println it }
println '==========================================='
foo.metaClass.properties.each { println it.name }
```

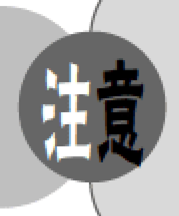
程序的输出为：

[1] 严格来说，不仅可以在 Groovy 类上使用 MOP，普通的 Java 类，甚至接口，也都可以使用。

```
public boolean java.lang.Object.equals(java.lang.Object)
public final native java.lang.Class java.lang.Object.getClass()
public native int java.lang.Object.hashCode()
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()
public java.lang.String java.lang.Object.toString()
public final void java.lang.Object.wait() throws java.lang.Interrupted-
Exception
public final native void java.lang.Object.wait(long) throws java.lang.
InterruptedException
public final void java.lang.Object.wait(long,int) throws java
.lang.InterruptedException
public java.lang.Object Foo.bar()
public groovy.lang.MetaClass Foo.getMetaClass()
public java.lang.Object Foo.getProp()
public java.lang.Object Foo.getProperty(java.lang.String)
public java.lang.Object Foo.invokeMethod(java.lang.String ,java.lang.
Object)
public void Foo.setMetaClass(groovy.lang.MetaClass)
public void Foo.setProp(java.lang.Object)
public void Foo.setProperty(java.lang.String,java.lang.Object)
==========================================
class
prop
metaClass
```

从上面的代码可以看到，通过 metaClass 可以轻松地对类的方法和属性进行遍历。通过 respondsTo 方法和 hasProperty 方法可以判断当前对象是否包含某一方法或属性。

```
class Foo{
   def prop
   def bar(){ }
}
def foo = new Foo()
assert foo.metaClass.respondsTo('bar')
assert foo.metaClass.hasProperty('prop')
assert ! foo.metaClass.respondsTo('foobar')
assert ! foo.metaClass.hasProperty('prop1')
```



respondsTo 方法和 hasProperty 方法不能识别类似上一节通过 invokeMethod 添加的未知方法。

17.3.2 动态添加方法

metaClass 可以为对象动态地添加方法：

```
class Foo{ }
def foo1 = new Foo()

//向 Foo 类添加方法：
Foo.metaClass.speak = {
    println 'foo speak'
}


def foo2 = new Foo()
foo2.speak()
//foo1.speak() 执行 foo1.speak()会报错，原因是找不到方法。
```

从上面的代码可以看出，在 Groovy 中可以直接向类的 metaClass 添加方法，具体方法为：

类名.metaClass.方法名 = 闭包

或者：

类名.metaClass.方法名 << 闭包

“=”可以覆盖已有的方法，“<<”表示加入新的方法。如果新加入的方法名称和参数都与旧的完全一致，使用“=”会覆盖旧的方法，而使用“<<”时会报错。

MOP 实现添加方法，实现得非常彻底，上面的方法名称也可以是动态的（与第 17.1 节相似，使用字符串），例如：

```
Foo.metaClass."${strMethodName}" = { }
```

如果动态添加的方法包含参数，则需要传入带参数的闭包：

```
Foo.metaClass.speak = { def text ->
    println "speak: ${text}"
}
def foo = new Foo()
foo.speak(1234)
foo.speak("hello world")
```

输出为：

```
speak: 1234
speak: hello world
```

如果闭包中不定义参数，则与上例相同，可以传入任意类型的参数。如果指定严格类型的参数，则调入时必须传入指定的类型，例如：

```
Foo.metaClass.speak = { int num ->
    println "speak: ${num}"
}
def foo = new Foo()
foo.speak(1234)
//foo.speak("hello world") 此时会报错，原因是方法未定义
```

如果希望添加强制无参数的方法，则闭包也需要定义为无参数的闭包：

```
Foo.metaClass.speak = { ->
    println "speak without params "
}
def foo = new Foo()
foo.speak()
//下面的两行会报错：
//foo.speak(1234)
//foo.speak("hello world")
```

通过 MOP 添加的方法，也支持重载（over load）特性：

```
Foo.metaClass.speak = { int num ->
    println "speak number: ${num}"
}
Foo.metaClass.speak = { String text ->
    println "speak text: ${num}"
}
foo.speak(1234)
foo.speak("hello world")
```

此时的输出为：

```
speak number: 12
speak text: hello world
```

通过 MOP 机制，还可以动态添加构造函数。只需要指定 metaClass 的 constructor 属性。可以使用“=”或者使用“<<”运算符传入一个闭包：

```
class Goods {
    String title
}
Goods.metaClass.constructor << { String title -> new Goods (title:title) }

//调用构造函数
def goods = new Goods("The Stand")
```

不过使用 MOP 添加构造函数要格外小心，因为有可能发生无穷递归，从而发生堆栈溢出的错误，例如：

```
class Goods {
    String title
}
Goods.metaClass.constructor << { new Goods () }

//调用构造函数
def goods = new Goods()
//此时，会发生死循环。闭包中的 new Goods ()会再次触发对闭包的调用，
//从而发生无穷递归。
```

使用 MOP 同样支持动态添加静态方法，其方法与加入普通方法非常相似，具体为：

类名.metaClass. **'static'**.方法名 = 闭包

或者：

类名.metaClass. **'static'**.方法名 << 闭包

例如：

```
class Goods {
   String title
}

Goods.metaClass.'static'.create << { String title -> new Goods (title:title) }

def goods = Goods.create("The Stand")
```

显然，通过 MOP 机制，可以轻松地实现动态添加方法。所以也不难想象，为什么 Domain 包中的类会自动地拥有持久化的方法。Groovy 在语言级别有了 MOP 这样的基础功能，那么用 Groovy 实现什么样的框架，就要看使用者的想象力了。使用 MOP，可以向别的类“借”方法。Groovy 中可以用&运算符将方法转换为闭包，因此，“借”方法不难实现：

```
class Foo {}
class Class2 {
    def method() { }
}
def c2 = new Class2()
//将其他对象的方法转换为闭包，然后就可以“借”了，不过借完是不需要还的
Foo.metaClass.newMethod = c2.&method
```

17.3.3　动态添加属性

使用 MOP 除了可以动态添加方法外，还可以添加属性，否则 Domain 类就不会被自动添加 id 和 version 了。

通过 MOP 添加属性是非常简单的，具体方法为：

类名.metaClass.属性名 = 属性值

这里的属性值比较重要，因为属性值的类型将决定属性的类型。给对象实例指定属性值时，如果使用了与属性值类型不符的变量，则会报错：

```
class Goods {}
Goods.metaClass.id = 1L //动态添加类型为 Long，默认值是 1，且名称为 id 的属性
def goods1 = new Goods(id:2L)
def goods2 = new Goods(id:3L)
assert goods1.id == 2L
assert goods2.id == 3L

//goods1.id = 'string' 这里会报错，原因是 id 是 Long 类型，不是 String 类型
```

除了上面的做法外，还可以通过添加方法实现添加属性。本质上，Java 和 Groovy 对象的属性都是由 setter 和 getter 两个方法实现的。因此，通过添加方法，也可以实现添加属性，例如：

```
class Goods {}
Goods.metaClass.getId = { 0L }
def goods = new Goods()
assert goods.id == 0L
```

然而，由于仅仅是添加方法，并没有在对象中添加成员变量，因此想添加可读可写属性并不容易，还需要设计如何存储对象数据：

```
//定义全局的、线程安全的属性 Map(ConcurrentHashMap),
//用于存储不同对象的不同的属性值
import java.util.concurrent.*
def properties = new ConcurrentHashMap()

Goods.metaClass.setId = { Long value ->
  properties[System.identityHashCode(delegate) + "id"] = value
}
Goods.metaClass.getId = {->
  properties[System.identityHashCode(delegate) + "id"]
}
```

第二种方法虽然也能实现添加属性，但显然不如第一种直观，所以一般也不推荐使用。

17.3.4　使用方法对象

熟悉 Java 反射的读者，一定对 Method 对象不会陌生，一个 Method 实例就对应为某个对象的成员方法。类似地，Groovy 的 MOP 中也包含类似的功能，不过更简单，也更友好，例如：

```
class Foo{
  def bar() {'MOP'}
```

```
}
Foo.metaClass.invokeMethod = { String name , args ->
    //注意方法的读取方式
    def metaMethod = delegate.metaClass.getMetaMethod(name, args)
    def result
if(metaMethod) {
    //这里需要注意后半句，如何调用方法对象
        result = 'Hello ' + metaMethod.invoke(delegate,args)
    } else {
        throw new MissingMethodException()
    }
    result
}
def foo = new Foo()
println foo.bar()
//println foo.speak() 执行这里会抛出异常
```

通过 metaClass 的 getMetaMethod()方法可以获取 metaMethod 对象，调用 metaMethod 对象的 invoke 方法，就可以执行相应的方法了。

17.3.5 为某一特定的实例添加方法

前面介绍的 MOP 技术都是“类”级别的。修改某一类的 metaClass，则之后用该类创建的实例都会有所改变。事实上，MOP 支持比类更细粒度的使用级别，可以单独在某一个对象上使用 MOP。

对于 Groovy 对象，可以直接创建 ExpandoMetaClass 实例，并使用它替换 Groovy 对象的 metaClass：

```
class Goods()
def g1 = new Goods()    // g 是 Groovy 对象，实现了 GroovyObject 接口
def g2 = new Goods()

def emc = new ExpandoMetaClass( g1.class)
emc.test = { println "测试" }
emc.initialize()

g1.metaClass = emc
g1.test()
//g2.test() 执行 g2.test()会报错，因为只有 g1 拥有 test 方法
```

这里要注意一点，不能将程序写成如下的形式：

```
g1.metaClass = new ExpandoMetaClass( gstr.class )
g1.metaClass.test = { println "test" }
```

原因是一旦替换了某一对象的 metaClass，则必须在其 metaClass(ExpandoMetaClass)

的 initialize()方法执行完毕之后，才能调用它的其他方法。以上面的代码为例，第一行替换了 g1 的 metaClass，则第二行代码会报错，原因是 g1 的 metaClass 在没有执行过 initialize()方法的前提下执行了 g1.metaClass.test = { … }，相当于调用了 g1.getMetaClass()方法。也就是说，它在没有调用过 initialize()方法的情况下，调用了 getMetaClass()方法，因而会报错。

关于 ExpandoMetaClass，还需要再补充一点：执行了 ExpandoMetaClass 的 initialize()方法之后，就不能再向 ExpandoMetaClass 添加方法了。通常向某一 Groovy 对象添加方法，都会写成这样的形式：

```
...
def emc = new ExpandoMetaClass( g1.class)
emc.test = { println "测试" }
emc.initialize()
g1.metaClass = emc
...
```

MOP 中同样也可以对单个的 Java 对象实例添加方法，但由于普通的 Java 类没有实现 GroovyObject 接口，所以实例中也不存在 metaClass 属性。因而向 Java 对象添加方法的方法稍微有所不同，需要用 groovy.util.Proxy 对 Java 对象进行包装：

```
def obj = new Pojo() //假设创建了不包含任何方法的 Java 类 Pojo

ExpandoMetaClass emc = new ExpandoMetaClass( obj.class, false )
emc.echo = { "向 Java 对象添加动态方法" }
emc.initialize()

//obj.metaClass = emc 此时给 obj 赋值会报错
obj = new groovy.util.Proxy().wrap( obj )
obj.setMetaClass(emc) //将 metaClass 应用于 obj
println obj.echo()
```

输出：

```
向 Java 对象添加动态方法
```

对单个的 Java 对象实例添加方法，是不是也很简单？

17.4 本章小结

本章对 Groovy 的部分高级特性进行了介绍，并重点阐述了 Groovy 的 MOP 机制。Grails 的神奇很大程度上源于 Groovy 强大的 MOP 机制。只有了解了 Groovy 的动态特性，才能深入学习 Grails 的原理和阅读 Grails 的源码。对这部分内容感兴趣的读者，不妨多花些时间阅读一下 Groovy 的官方网站。如果能够打下扎实的 Groovy 基础，阅读 Grails 源码会事半功倍。

第18章 Grails 插件开发

第 16 章介绍了 Grails 插件的安装和使用，本章将讨论 Grails 插件开发相关的技术。开发 Grails 的插件，需要掌握 Groovy 的 MOP 编程技术。对于还没有掌握 MOP 技术的读者应先阅读第 17 章高级 Groovy 特性，补充一下这方面的知识。

本质上，Grails 各个模块的大部分功能都是通过系统插件的形式组织的：URL mappings、codecs、controllers、core、dataSource、domainClasses、filters、hibernate、i18n、logging、scaffolding、services、servlets、Web flow 等各个模块，是分别使用了不同的插件来实现的。与其他插件唯一不同的是，它们是以系统插件的形式存在的，无需单独安装。

18.1 创建与发布插件

Grails 中提供了简便的插件创建命令，create-plugin[1]：

```
grails create-plugin MyPlugin
```

此时，Grails 会在项目的根目录创建一个名为 MyPlugin 的文件夹，其内部的文件组织结构与典型的 Grails 项目是非常相似的：

```
GDepot\MyPlugin\src
GDepot\MyPlugin\src\java
GDepot\MyPlugin\src\groovy
GDepot\MyPlugin\grails-app
GDepot\MyPlugin\grails-app\controllers
GDepot\MyPlugin\grails-app\services
GDepot\MyPlugin\grails-app\domain
GDepot\MyPlugin\grails-app\taglib
GDepot\MyPlugin\grails-app\utils
GDepot\MyPlugin\grails-app\views
GDepot\MyPlugin\grails-app\views\layouts
GDepot\MyPlugin\grails-app\i18n
GDepot\MyPlugin\grails-app\conf
GDepot\MyPlugin\test
```

[1] create-plugin 命令不需要在某一 Grails 工程内运行，因而它创建的插件不与任何具体的 Grails 工程相关。可以将新创建的插件理解为是一个全新的工程，即插件工程。

```
GDepot\MyPlugin\test\unit
GDepot\MyPlugin\test\integration
GDepot\MyPlugin\scripts
GDepot\MyPlugin\web-app
GDepot\MyPlugin\web-app\js
GDepot\MyPlugin\web-app\css
GDepot\MyPlugin\web-app\images
GDepot\MyPlugin\web-app\META-INF
GDepot\MyPlugin\lib
GDepot\MyPlugin\grails-app\conf\spring
GDepot\MyPlugin\grails-app\conf\hibernate
GDepot\MyPlugin\web-app\WEB-INF
GDepot\MyPlugin\web-app\WEB-INF\tld
```

插件工程与普通 Grails 工程相似，运行插件也和运行普通 Grails 工程完全一致。需要进入工程文件夹（MyPlugin）并运行 run-app 命令：

```
>grails run-app
```

同理，插件工程也可以进行自动化测试：

```
>grails test-app
```

插件文件夹的根目录中有一个非常重要的 Grails 插件描述文件。按照约定优于配置的原则，其文件名为“插件名+GrailsPlugin.groovy”即 MyPluginGrailsPlugin.groovy。通过这个类，可以配置插件的描述信息，包括版本、作者信息等：

```
class MyPluginGrailsPlugin {
    def version = 0.1
    def dependsOn = [:]

    // TODO Fill in these fields
    def author = "Your name"
    def authorEmail = ""
    def title = "Plugin summary/headline"
    def description = '''\
     Brief description of the plugin.
     '''
     ...
}
```

其中：

（1）title——插件的标题；

（2）author——插件作者的姓名；

（3）authorEmail——插件作者的 E-mail；

（4）description——多行字符串，用于描述插件的用途；

（5）documentation——插件文档的 URL。

当某一插件开发完毕后，需要使用 package-plugin 命令对插件进行打包：

```
>grails package-plugin
```

Grails 会创建名为“grails-插件名称-版本号.zip”的压缩包。但压缩包中不会包含插件工程的全部内容，以下部分会被排除在外：

（1）grails-app/conf/DataSource.groovy；

（2）grails-app/conf/UrlMappings.groovy；

（3）web-app/WEB-INF/*.*。

这里需要说明的是，如果需要在插件内定义 Url 映射，则需要给配置文件起一个全新的名称，例如 MyPluginUrlMappings.groovy。重新命名的文件不会被排除在外。

如果希望将自己开发的插件提交给 Grails 官方并向全世界的 Grails 开发人员共享，可以通过 http://www.g2one.com 联系 G2One team，他们审核通过并分配相应的权限。此后，使用 Grails 命令 release-plugin 就可以将插件的更新同步发布到 Grails 官方的源代码管理容器中。相应地，执行 list-plugin 命令时看到的返回结果也会包含更新。

18.2 插件能做什么

“插件能做什么？”似乎任何 Grails 插件开发相关的内容都应该从这个问题开始入手。从上一节讨论的 Grails 的插件组织结构可以了解到，Grails 的插件工程实际上和一个普通的 Grails 工程非常相似。因而，在插件项目中，可以和普通的 Grails 项目一样，创建 Controller、GSP 页面、Domain 类、Service、自定义 Taglib、自定义编码（Codec）、自定义控制台命令等。

除了这些基本的功能外，还可以通过插件在其他几个方面进行开发。回忆一下 MyPluginGrailsPlugin 类的内容：

```
class MyPluginGrailsPlugin {
    def version = 0.1
    def dependsOn = [:]

    // TODO Fill in these fields
    def author = "Your name"
    def authorEmail = ""
    def title = "Plugin summary/headline"
    def description = '''\
     Brief description of the plugin.
     '''
    // URL to the plugin's documentation

    def documentation = "http://grails.org/MyPlugin+Plugin"
```

```
    def doWithSpring = {

     }

    def doWithApplicationContext = { applicationContext ->

    }

    def doWithWebDescriptor = { xml ->

    }

    def doWithDynamicMethods = { ctx ->

    }

    def onChange = { event ->

    }

    def onConfigChange = { event ->

    }
}
```

除了编写用于描述插件信息的属性外，还有 6 个闭包，分别对应若干功能，分别是：添加 Spring 配置信息、与 Spring ApplicationContext 交互、参与生成 web.xml、添加动态方法、捕获程序和配置信息的变更。正是这些看似平凡的功能，构成了实现 Grails 的基础。下面对这几点逐一加以讨论。

在讨论插件的各个功能点之前，不妨先了解一下插件类的属性，每个插件类隐式地包含了 application 属性。application 是 GrailsApplication 接口的实例。GrailsApplication 包含了很多非常实用的方法。

（1）*Classes——动态方法，获取系统中特定类型的类，如 application.controllerClasses、application.domainClasses。

（2）get*Class——动态方法，通过类名获取某一符合特定类型的类，如 application.getControllerClass('GoodsController')。

（3）is*Class——动态方法，判断某一类是否为某一类型，如：application.isControllerClass(GoodsController.class) 。

（4）add*Class——动态方法，将某一类加入到系统中，如：application.addControllerClass(GoodsController.class)。

这里还需要注意一点，*Classes 和 is*Class 返回的是 GrailsClass 接口的实例（或者 GrailsClass 接口的数组），例如：

```
application.domainClasses.each { dc ->
    //这里的 dc 是 GrailsClass 类型
    println dc.name
}
```

GrailsClass 也提供了大量实用的方法。

（1）getPropertyValue——读取属性的初始值。

（2）hasProperty——判断是否包含某一属性。

（3）newInstance——创建当前类的新实例。

（4）getName——返回类的逻辑名称（不包括包名以及约定部分的名称，例如，GoodsController 将返回 Goods）。

（5）getShortName——返回不包含前缀的类名。

（6）getFullName——返回完整的类名。

（7）getNaturalName——返回更友好的名称（例如，'lastName' 会变成 'Last Name'）。

（8）getPackageName——返回包名。

18.2.1 添加 Spring 配置信息

doWithSpring 闭包用于向 Spring 添加配置信息。以 Grails 自带的 Hibernate 插件为例，在启动时需要先向 Spring 容器添加更多的 bean，其中向 Spring 容器中添加 bean 的方法可以参考第 12.2 节介绍的 Spring DSL：

```
def doWithSpring = {
    ...
    application.domainClasses.each {dc ->
        "${dc.fullName}Validator"(HibernateDomainClassValidator) {
        messageSource = ref("messageSource")
        domainClass = ref("${dc.fullName}DomainClass")
    }
    ...
    sessionFactory(ConfigurableLocalSessionFactoryBean) {
        dataSource = dataSource
        if (application.classLoader.getResource("hibernate.cfg.xml")) {
            configLocation = "classpath:hibernate.cfg.xml"
        }
        if (hibConfigClass) {
            configClass = ds.configClass
        }
        hibernateProperties = hibernateProperties
        grailsApplication = ref("grailsApplication", true)
        lobHandler = lobHandlerDetector
        entityInterceptor = eventTriggeringInterceptor
    }
```

```
        transactionManager(HibernateTransactionManager) {
            sessionFactory = sessionFactory
        }
        ...
    }
```


上面的代码部分引自 Grails Hibernate 插件的源程序，使用的 Spring DSL 在 Spring 容器中定义了 Domain 类的 Validator、SessionFactory、transactionManager 等。

类似地，Acegi 插件也在 doWithSpring 闭包中添加了大量的 Spring 配置信息：

```
    def doWithSpring = {
    ...
        openIDAuthProvider(OpenIDAuthenticationProvider) {
            userDetailsService = ref('userDetailsService')
        }

        openIDConsumerManager(ConsumerManager) {
            nonceVerifier = ref('openIDNonceVerifier')
        }
    ...
    }
```

通过上面两个例子，可以明确 doWithSpring 闭包的用途：在 doWithSpring 闭包中编写 Spring 的 DSL，从而使插件能够参与配置 Spring 容器中的 bean。

18.2.2 与 Spring 容器交互

doWithApplicationContext 闭包也用于编写初始化 Spring 的代码。与 doWithSpring 不同的是，doWithApplicationContext 闭包是在 Spring 容器初始化完成之后被调用的，同时，该闭包中可以访问 Spring 容器的 ApplicationContext。通过 ApplicationContext 对象，可以容易地访问 Spring 容器中的 Bean。以 Grails 中用于国际化支持的 I18nGrailsPlugin 插件为例，它需要在完成 Spring 容器的初始化后，对 messageSource.resourceLoader 进行配置。

```
    def doWithApplicationContext = { ctx ->
        if(!application.warDeployed) {
            ctx.messageSource.resourceLoader  = ctx.messageSourceLoader
        }
    }
```

18.2.3 修改 web.xml

Grails 的准则是约定优于配置，因此默认情况下不允许修改 web.xml 文件。但是理想总归是理想，web.xml 在 JSP\Servlet 技术中有着非常重要的地位，Servlet、Filter、Listener

等元素，都需要通过 web.xml 进行配置。因而，在实际项目中很有可能需要修改该文件。考虑到这一问题，Grails 允许通过插件对 web.xml 进行配置，可以根据需要，在插件的 doWithWebDescriptor 闭包中编写 Groovy 针对 XML 的 DSL。其中，闭包的参数是通过 Groovy 的 XmlSlurper 类解析 XML 文件时返回的 GPathResult 对象。

以 Grails 的系统插件 ControllersPlugin 为例：

```
def doWithWebDescriptor = { webXml ->
    def mappingElement = webXml.'servlet-mapping'
    mappingElement + {
        'servlet-mapping' {
            'servlet-name'("grails")
            'url-pattern'("*.dispatch")
        }
    }
}
```

ControllersPlugin 通过 doWithWebDescriptor，添加了 servlet 映射。

Acegi 插件同样也修改了 web.xml 的内容：

```
def doWithWebDescriptor = { xml ->
    def conf = getSecurityConfig()
    if (conf && conf.active) {
        def contextParam = xml.'context-param'
        contextParam[contextParam.size() - 1] + {
            'filter' {
                'filter-name'('springSecurityFilterChain')
                'filter-class'(DelegatingFilterProxy.name)
            }
        }
        def filter = xml.'filter'
        filter[filter.size() - 1] + {
            'filter-mapping'{
                'filter-name'('springSecurityFilterChain')
                'url-pattern'('/*')
            }
        }
    }
}
```

如果需要深入了解 GPathResult 对象的使用，可进一步参考 Groovy 的官方网站：http://groovy.codehaus.org/Reading+XML+using+Groovy%27s+XmlSlurper。

在 Grails 的插件列表中，还有专门用于修改 web.xml 的插件，即 WebXML。使用该插件可以更容易定置 web.xml 的内容。若要了解更多详情，可进一步参考 http://www.grails.org/WebXML+Plugin。

18.2.4 添加动态方法

doWithDynamicMethod 闭包可以实现添加动态方法。闭包的参数是 Spring 的 ApplicationContext，因此添加的动态方法可以使用 Spring 容器中的资源。Grails 的 Hibernate 插件，通过添加动态方法，使得 Domain 类都包含了持久化相关的方法。而添加动态方法的原理在上一章已经讨论过了，是通过 Groory 语言的 MOP 机制实现的，例如：

```
class ExamplePlugin {
  def doWithDynamicMethods = { applicationContext ->
        application.controllerClasses.each { controllerClass ->
              controllerClass.metaClass.myNewMethod = {->
                  println "hello world"
              }
        }
  }
}
```

此时，所有的 Controller 类中都可以调用 myNewMethod 方法。实际的 Grails 插件也大量使用了 doWithDynamicMethod 闭包。Hibernate 插件的代码稍显复杂，不感兴趣的读者不妨跳过，这里仅部分摘录其内容如下：

```
def doWithDynamicMethods = {ctx ->
    SessionFactory sessionFactory = ctx.sessionFactory
    def lazyInit = {GrailsDomainClass dc ->
        registerDynamicMethods(dc, application, ctx)
        for (subClass in dc.subClasses) {
            registerDynamicMethods(subClass, application, ctx)
        }
        MetaClass emc =
            GroovySystem.metaClassRegistry.getMetaClass(dc.clazz)
    }

    for (dc in application.domainClasses) {
        //   registerDynamicMethods(dc, application, ctx)
        MetaClass mc = dc.metaClass
        def initDomainClass = lazyInit.curry(dc)
        def findAllMethod = new FindAllPersistentMethod(sessionFactory,
        application.classLoader)
        mc.'static'.findAll = {->
            findAllMethod.invoke(mc.javaClass, "findAll", [] as Object[])
        }
        ...
        mc.methodMissing = { String name, args ->
            initDomainClass()
```

```
                mc.invokeMethod(delegate, name, args)
            }
            mc.'static'.methodMissing = {String name, args ->
                initDomainClass()
                def result
                if (delegate instanceof Class) {
                    result = mc.invokeStaticMethod(delegate, name, args)
                } else {
                    result = mc.invokeMethod(delegate, name, args)
                }
                result
            }
            addValidationMethods(dc, application, ctx)
        }
    }
```

Hibernate 插件添加动态方法的过程非常巧妙。为了避免启动时间过长，设计了延时加载的机制。只有当 Domain 类发生 missing method 事件时，才调用 initDomainClass，以实现向 Domain 类添加动态方法。

18.2.5 捕获变更

用 Grails 开发 Web 应用的时候，通常修改程序时不需要重新启动应用，这使得开发体验变得更加友好。然而，计算机中不存在神奇，一切都是程序控制的。这后面又是通过怎样的机制，实现修改程序而无需重启服务器的呢？

Grails 的插件类中提供两个闭包 onChange 和 onConfigChange，它们分别用于捕获源程序的变更和捕获配置文件的变更，从而可以实现在变更发生时，重新读取相应的程序或资源。

onChange 和 onConfigChange 闭包的参数是 event 对象，该对象包含以下几个实用的属性。

（1）event.source——事件来源，可能是类文件，也可能是 Spring 资源文件；

（2）event.ctx——Spring 的 ApplicationContext 对象；

（3）event.plugin——管理资源的插件对象实例，通常是 this；

（4）event.application——GrailsApplication 实例。

以系统插件 ServicePlugin 为例，该插件需要监视 Service 类的变化，当 Service 类发生改变时，将变化的类重新装入系统，并更新 Spring 容器：

```
class ServicesGrailsPlugin {
    ...
    def watchedResources = "file:./grails-app/services/*Service.groovy"
    ...
    def onChange = { event ->
```

```
            if(event.source) {
                def serviceClass = application.addServiceClass(event.source)
                def serviceName = "${serviceClass.propertyName}"
                def beans = beans {
                    "$serviceName"(serviceClass.getClazz()) { bean ->
                        bean.autowire =  true
                    }
                }
                if(event.ctx) {
                    event.ctx.registerBeanDefinition(serviceName,
                        beans.getBeanDefinition(serviceName))
                }
            }
        }
    }
```

其中，watchedResources 属性用于定义被监视的资源。watchedResources 属性既可以是 String 类型，也可以是 List 类型。

Hibernate 插件的 onChange 事件处理得更彻底，直接重启了应用：

```
def onChange = {event ->
    if (event.source instanceof Resource) {
        restartContainer()
    }
}
```

除了程序、资源的变更外，Grails 也为配置文件的变更设置了相应的处理事件。onConfigChange 闭包用于捕获配置文件发生的变化。以系统插件 LoggingGrailsPlugin 为例：

```
def onConfigChange = {event ->
    def log4jConfig = event.source.log4j
    if (log4jConfig) {
        def props = log4jConfig.toProperties('log4j')
        log.info "Updating Log4j configuration.."
        def resourcesDir=System.getProperty(
            GrailsApplication.PROJECT_RESOURCES_DIR)
        if(resourcesDir) {
            new File("$resourcesDir/log4j.properties").withOutputStream
            {out ->
                props.store(out, "Grails' Log4j Configuration")
            }
        }
    }
}
```

关于捕获变更，还存在一个需求，不同的插件有可能存在依赖。某一插件捕获了变更，需要通知其他插件更新，或者某一插件需要知道是否其他插件发生了更新。Grails 的插件系统对这两种情况都提供了支持。

observe 属性用于定义“观察”其他插件的变化。如果所观察的插件发生了变化，则也需要触发变更事件。例如，观察 Hibernate 插件是否发生了变化：

```
def observe = ["hibernate"]
```

influences 属性用于通知其他插件处理变化。例如，如果当前插件的变化需要通知 Controller 插件，则应编写如下代码：

```
def influences = ['controllers']
```

18.3 插件的依赖关系

插件与插件之间是存在依赖关系的，例如 Hibernate 插件依赖于 dataSource 插件，也一定是依赖于 Domain 插件的。如果 dataSource 插件和 Domain 插件没有启动，Hibernate 插件显然也是无法启动的。

使用 dependsOn 属性，可以配置插件间的依赖关系。dependsOn 属性是 Map 类型，Map 的键是插件名称，值是版本号：

```
def version = grails.util.GrailsUtil.getGrailsVersion()
def dependsOn = [dataSource: version, i18n: version,
                 core: version, domainClass: version]
```

version 的写法还可以更加智能，例如：

```
def dependsOn = [MyPlugin : "* > 1.0"]
def dependsOn = [MyPlugin:"1.0 > 1.1"]
def dependsOn = [MyPlugin:"1.0 > *"]
```

其中“*”表示任意的版本，即“* > 1.0”表示大于 1.0 版的任意版本；“1.0 >*”表示小于 1.0 的任意版本。此外，当前的表达式在判断时会自动过滤掉 BETA、ALPHA 等后缀，因而“1.0 > 1.1”可以匹配以下的所有版本：1.1、1.0、1.0.1、1.0.3-SNAPSHOT、1.1-BETA2。

dependsOn 属性定义了比较严格的依赖关系，除了 dependsOn 外，还有一种方式可以定义相对较弱的依赖关系，即 loadAfter 属性，它用于仅指定载入的顺序。例如，Hibernate 插件虽然不严格依赖于 Controller 插件，但需要在启动 Controller 插件之后启动，因此它使用了 loadAfter 属性：

```
def loadAfter = ['controllers']
```

18.4 在安装或升级时执行附加操作

每个 Grails 项目和插件项目中，都包含名为 scripts 的文件夹，其中的 groovy 程序是自定义的控制台命令。

Grails 的众多控制台命令本质上是在运行 Apache Ant 的 Task。使用 Groovy 的 GAnt 技术，可以使用 Groovy 程序编写 Ant 的 build.xml[13][14]。

插件项目的 scripts 文件夹中默认包含了两个命令文件：_Intall.groovy 和_Upgrade.groovy，它们分别用于在安装和升级插件时，执行附加操作。

```
Ant.mkdir(dir:"${basedir}/grails-app/jobs")
Ant.copy(file:"${pluginBasedir}/src/samples/SamplePluginConfiguration.
groovy",todir:"${basedir}/grails-app/conf")
...
```

18.5 本章小结

本章介绍了 Grails 的插件开发技术。通过学习本章内容，可以开发用于 Grails 的插件，从而实现对 Grails 的自由扩展。Grails 的插件技术也是 Grails 的基础，整个 Grails 框架都需要运行在这一插件平台之上。理解 Grails 的插件原理，是学习 Grails 原理必要的前提。下一章将要讨论 Grails 源程序，本章的系统插件部分是重要的突破口。

第19章 浅析 Grails 的源程序

本章不打算对 Grails 的源码进行全面的分析，只是对其中几个功能点进行简单分析：

（1）HibernateCriteriaBuilder 一直被认为是很神奇的东西，本章将分析其实现原理，解密神奇的本质；

（2）第 11 章曾经提到过，使用 Grails 内置方法构造的数据库查询，都没有开启 Query-Cache，本章将尝试修改其源码，为查询添加能够配置 Query-Cache 的选项。

19.1 准备工作

19.1.1 下载源码

首先从 Grails 官方网站（http://www.grails.org/Download）下载 Grails1.0.4 的源代码（选择 **Source Zip** 或者 **Source Tar/GZ**），然后解压至本地硬盘。事实上，配置 Grails 环境变量时，就可以选用 **Source** 包，因为 **Source** 包中含有 **Binary** 包的全部内容。

19.1.2 编译 Grails 源码

可以看到源码中包含了如图 19-1 所示的内容。

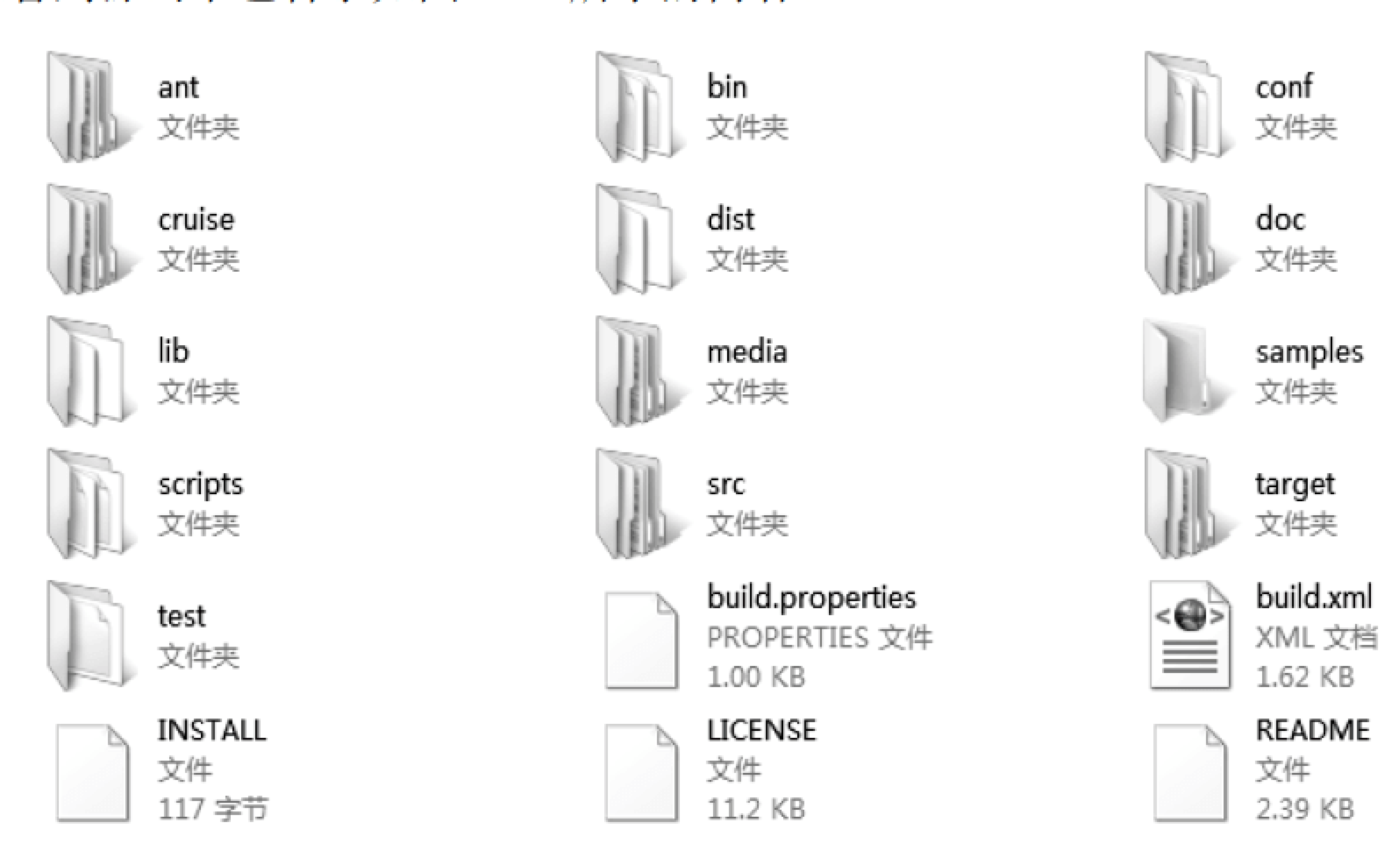


图 19-1 Grails 源码目录

其中的 ant 文件夹和 build.xml 会让用户眼前一亮。可以使用 ant 对 Grails 进行编译和打包。在控制台中输入：

```
>ant\bin\ant
```

该命令将运行构建 Grails 的 ant 脚本。但会有一个问题，该脚本的默认 task 需要花费非常长的时间用于自动测试，而且，如果使用 Java6 还可能出现测试失败的情况：

```
run-test:
   [delete] Deleting directory D:\grails-1.0.4\target\test-reports
   [mkdir] Created dir: D:\grails-1.0.4\target\test-reports
   [echo] Test being run * from target/test-classes
   [junit] TEST grails.build.eclipse.EclipseClasspathTests FAILED
```

阅读 ant 脚本 ant\build\unit-test.xml 可知，如果在配置文件中定义配置项为 skipTests=true，则构建时不会再执行自动测试。因此可以修改 build.properties 文件，加入 skipTests=true[1]，并重新运行 ant，可见如下结果：

```
BUILD SUCCESSFUL
Total time: 3 minutes 52 seconds
```

这一次可以看到提示构建成功的字样。但由于在构建过程中重新生成了文档，重新进行了打包（binary 包、source 包等），总时间还是显得过长。事实上，只需要重新编译和生成 jar 包就可以了。再次阅读 build.xml，default 任务需要依赖于 clean、build、test-with-coverage、jar、javadoc、package 任务。直接运行 jar 任务效果如何？可进行如下试验，直接在控制台输入相关命令并分析结果：

```
>ant\bin\ant jar
Buildfile: build.xml

build-init:

build:

build:checkj5:
     [echo] Building Java 5 code for Java version: 1.6

build:java5:

build-test:
  [groovyc] Compiling 16 source files to D:\grails-1.0.4\target\test-classes
    [javac] Compiling 17 source files to D:\grails-1.0.4\target\test-classes
Trying to override old definition of task groovyc
```

[1] 按道理讲，关闭自动测试存在风险，但 TEST grails.build.eclipse.EclipseClasspathTests FAILED 不是真正测试出了 bug，这个问题只存在于使用 Java 6 构建的系统。

```
build:checkj5:
     [echo] Building Java 5 code for Java version: 1.6

build-test:java5:
     [echo] Building Java 5 Test Sources

jar:
      [jar] Building jar: D:\grails-1.0.4\dist\grails-test-1.0.4.jar
      [jar] Building jar: D:\grails-1.0.4\dist\grails-spring-1.0.4.jar
      [jar] Building jar: D:\grails-1.0.4\dist\grails-cli-1.0.4.jar
      [jar] Building jar: D:\grails-1.0.4\dist\grails-core-1.0.4.jar
      [jar] Building jar: D:\grails-1.0.4\dist\grails-gorm-1.0.4.jar
      [jar] Building jar: D:\grails-1.0.4\dist\grails-web-1.0.4.jar
      [jar] Building jar: D:\grails-1.0.4\dist\grails-webflow-1.0.4.jar
      [jar] Building jar: D:\grails-1.0.4\dist\grails-crud-1.0.4.jar

BUILD SUCCESSFUL
Total time: 7 seconds
```

可以看出，构建脚本同样执行了编译操作，然后生成了 jar 包，而且速度非常快，只花了几秒钟。

如果环境变量中配置 GRAILS_HOME 指向的文件夹，就是当前这个运行 ant 脚本的文件夹，则构建后无需配置任何内容，否则，需要用新创建的 jar 包替换%GRAILS_HOME%\dist 文件夹下对应的 jar 包。

19.2 HibernateCriteriaBuilder 的原理

回顾第 5 章，相信读者一定对 HibernateCriteriaBuilder 的威力留下了深刻的印象。它是如此的神奇，使得数据库的查询变得如此简单。本节就对 HibernateCriteriaBuilder 的实现进行解密。

通过阅读 Hibernate 插件的源程序(src\groovy\org.codehaus\groovy\grails\plugins\orm\hibernate\HibernatePlugin.groovy)可以找到为 Domain 类添加 createCriteria 方法的源码：

```
metaClass.'static'.createCriteria = {->
    new HibernateCriteriaBuilder(domainClassType, sessionFactory)
}
```

可以看出，调用 Domain 类的 createCriteria 方法，实际上是创建并返回了 HibernateCriteriaBuilder 对象。通过源码还可以进一步确定，HibernateCriteriaBuilder 是一个 Java 类，其位于 src\persistence\grails\orm\HibernateCriteriaBuilder.java。

已经有些奇怪了，17 章介绍的高级 Groovy 特性，可以通过重载 invokeMethod 方法实现 DSL 的开发。既然 Groovy 可以很容易地实现 DSL，那么为什么 HibernateCriteriaBuilder 是一个 Java 类呢？

道理很简单，Java 在执行性能上比 Groovy 有优势。为了提高 Grails 的性能，整个 Grails 框架大概有 70%的源代码是使用 Java 开发的。现在的问题是，如何使用 Java 实现 DSL 呢？

事实上，Java 和 Groovy 是不分家的。当 Java 类实现了 GroovyObject 接口后，就拥有了一些 Groovy 的特性了[1]（如 invokeMethod）。HibernateCriteriaBuilder 派生于 GroovyObjectSupport 类，而 GroovyObjectSupport 类实现了 GroovyObject 接口：

```
public class HibernateCriteriaBuilder extends GroovyObjectSupport{
...
}
```

GroovyObjectSupport 类包含了如下的方法：

```
MetaClass getMetaClass()——返回当前对象的 metaclass
Object getProperty(String property)——返回给定属性的属性值
Object invokeMethod(String name, Object args)——调用指定的方法
void setMetaClass(MetaClass metaClass)——设定新的 metaClass
void setProperty(String property, Object newValue)——设置给定属性的属性值
```

有了 invokeMethod 方法，现在若想使用 Java 实现 DSL，就变得和 Groovy 一样容易了。其核心思想是，在对象内部构造 Hibernate 的 Criteria 对象，然后通过执行闭包向 Criteria 对象中添加查询条件。

```
public Object invokeMethod(String name, Object obj) {
    //取参数
    Object[] args = obj.getClass().isArray() ?
    (Object[])obj : new Object[]{obj};
    //
    if(isCriteriaConstructionMethod(name, args)) {
        if(this.criteria != null) {
            throwRuntimeException(
                new IllegalArgumentException("call to [" + name
                + "] not supported here"));
        }
        ...
        boolean paginationEnabledList = false; //从这里可以看到,
                                               //支持分页是可能的
        createCriteriaInstance();              //创建 Criteria 对象
        // Check for pagination params
        if(name.equals(LIST CALL) && args.length == 2) {
            //当方法名为 list 且有两个参数（一个是闭包，一个是分页 Map）
```

[1] 严格地说，是 Java 类实现了 GroovyObject 接口后，如果在 Groovy 中调用，才具有 Groovy 的特性。

```
                //则进行分页查询
                paginationEnabledList = true;
                invokeClosureNode(args[1]);
            } else {
                invokeClosureNode(args[0]);
            }
            ...
            Object result;
            if(!uniqueResult) {
                if(scroll) { //scorll 方法
                    result = this.criteria.scroll();
                }
                else if(count) { // count 方法
                    this.criteria.setProjection(Projections.rowCount());
                    result = this.criteria.uniqueResult();
                } else if(paginationEnabledList) { //分页的 list 方法
        // Calculate how many results there are in total. This has been
        // moved to before the 'list()' invocation to avoid any "ORDER
        // BY" clause added by 'populateArgumentsForCriteria()',
        // otherwise
        // an exception is thrown for non-string sort fields (GRAILS-2690).
                    this.criteria.setFirstResult(0);
                    this.criteria.setMaxResults(Integer.MAX_VALUE);
                    this.criteria.setProjection(Projections.rowCount());
                    //取出记录总数
                    int totalCount =
                  ((Integer)this.criteria.uniqueResult()).intValue();
                    // Drop the projection, add settings for the pagination
                    // parameters, and then execute the query.
                    this.criteria.setProjection(null);
                    this.criteria.
                    setResultTransformer(
                        CriteriaSpecification.ROOT_ENTITY);
                    GrailsHibernateUtil.
                        populateArgumentsForCriteria(
                            this.criteria, (Map)args[0]);
                    PagedResultList pagedRes = new
                        PagedResultList(this.criteria.list());

        // Updated the paged results with the total number of records
        // calculated previously.
                    pagedRes.setTotalCount(totalCount);
                    result = pagedRes;
                } else { //不分页的 list 方法
                result = this.criteria.list();
```

```
                }
            }
            else {
                result = this.criteria.uniqueResult();
            }
            if(!this.participate) {
                this.session.close();
            }
            return result;
        }
        else {
            //其他，如 and、or 等
            ...
        }
        closeSessionFollowingException();
        throw new MissingMethodException(name, getClass(), args) ;
    }
```

从上面摘录的代码中，可以看到 HibernateCriteriaBuilder 支持分页查询。但不知道为什么没有在文档中公开。

有两个方法比较重要，分别是：invokeClosureNode()方法，用于调用闭包；GrailsHibernateUtil.populateArgumentsForCriteria()方法，用于将分页 Map 的内容转换为 Criteria 中的查询逻辑。

```
    private void invokeClosureNode(Object args) {
        Closure callable = (Closure)args;
        callable.setDelegate(this);
        callable.call();
    }
```

invokeClosureNode 方法用于调用闭包，注意“**callable.setDelegate(this);**”是实现这一 DSL 的关键。这一行的代码使得闭包中的程序可以调用 HibernateCriteriaBuilder 的成员方法。例如，查询时使用的 eq、ge、le、like 等方法，都是 HibernateCriteriaBuilder 的成员方法，可以直接在闭包中调用。

```
    //闭包中调用的 eq 方法，本质上调用了 HibernateCriteriaBuilder 的 eq 方法
    public Object eq(String propertyName, Object propertyValue) {
        if(!validateSimpleExpression()) {
            throwRuntimeException( new IllegalArgumentException(
                "Call to [eq] with propertyName ["+
                propertyName+"] and value ["+
                propertyValue+"] not allowed here."));
        }
        propertyName = calculatePropertyName(propertyName);
        propertyValue = calculatePropertyValue(propertyValue);
        return addToCriteria(Restrictions.eq(propertyName, propertyValue));
```

```
}
```

关于 GrailsHibernateUtil.populateArgumentsForCriteria()方法的实现细节，参见下一节。

之所以选择 HibernateCriteriaBuilder，不仅仅是因为它的神奇和有趣。Grails 提供了大量的 DSL（Builder），而这些 DSL 的实现原理基本上是一致的。理解了 HibernateCriteriaBuilder，再去学习诸如 Domain 的 mapping、Spring Integration、Web Flow 等的原理，都会变得比较轻松。

19.3 开启 Hibernate Query Cache

Hibernate Query Cache 是用查询数据库 SQL 语句作为 cache 的键，缓存查询结果的主键集合。使用 Query Cache 技术，可以实现对相同的 SQL 不重复地查询数据库，从而减小开销并提高性能。

Grails 封装的查询方法都无法开启 Query Cache，因此目前只能通过 HibernateCriteriaBuilder 使用 Query Cache。Grails 社区已经认识到了这一点，计划在分页 Map 中加入 cache 键，当 cache 键值为 true 且查询的 Domain 类配置了二级 Cache，则执行查询时使用 Query Cache。

开源的好处就在于，用户可以自己动手对其进行修改。要实现上面提出的功能，只需要解决两个问题：第一，判断某一个 Domain 类是否使用了 cache；第二，判断分页 Map 中是否包含 cache 键，并根据它的值，设置 Query Cache。

首先，解决第一个问题。通过 org.codehaus.groovy.grails.orm.hibernate.cfg 包中的 GrailsDomainBinder 类，可以读取 Domain 类的 mapping 信息。

```
Mapping m = GrailsDomainBinder.getMapping(clazz)
```

其中，clazz 参数是具体 Domain 实例的 class 属性，返回值的 Mapping 类是 org.codehaus.groovy.grails.orm.hibernate.cfg 包下面的一个 Groovy 类。Mapping 类中包含了 cache 属性，类型为 CacheConfig。CacheConfig 类和 Mapping 类在同一个包中，代码如下：

```
class CacheConfig {
    static final USAGE_OPTIONS =
     ['read-only','read-write','nonstrict-read-write','transactional']
    static final INCLUDE_OPTIONS = ['all', 'non-lazy']

    String usage = "read-write"
    boolean enabled = false
    String include = "all"
}
```

显然，这个 enabled 属性正是判断 Domain 类是否开启了二级缓存所需要的。如果使用 Groovy 进行判断，则应写成如下样式：

```
GrailsDomainBinder.getMapping(domainClazz)?.cache?.enabled
```

如果用 Java 实现，则稍显复杂：

```
Mapping mapping = GrailsDomainBinder.getMapping(domainClazz);
return mapping != null && mapping.getCache() != null &&
        mapping.getCache().getEnabled();
```

不妨在 GrailsHibernateUtil 类中添加一个静态方法：

```
public static boolean isDomainCacheable( Class domainClazz) {
    Mapping mapping = GrailsDomainBinder.getMapping(domainClazz);
    return mapping != null && mapping.getCache() != null &&
        mapping.getCache().getEnabled();
}
```

至此，判断 Domain 类是否开启了二级缓存的问题已经解决了。接下来，实现在分页 Map 中添加 cache 键，并根据该键值配置 Query Cache。GrailsHibernateUtil 类中包含了名为 populateArgumentsForCriteria 的方法，它会根据 Map 的内容，将分页逻辑、排序等约束，添加到 Criteria 中。

```
public static void populateArgumentsForCriteria(Criteria c, Map argMap) {
    Integer maxParam = null;
    Integer offsetParam = null;
    if(argMap.containsKey(ARGUMENT_MAX)) {
        maxParam =
         (Integer)converter.convertIfNecessary(argMap.get(ARGUMENT_MAX),
         Integer.class);
    }
    if(argMap.containsKey(ARGUMENT_OFFSET)) {
        offsetParam =
     (Integer)converter.convertIfNecessary(
         argMap.get(ARGUMENT_OFFSET),Integer.class);
    }
    String orderParam = (String)argMap.get(ARGUMENT_ORDER);
    Object fetchObj = argMap.get(ARGUMENT_FETCH);
    if(fetchObj instanceof Map) {
        Map fetch = (Map)fetchObj;
        for (Iterator i = fetch.keySet().iterator(); i.hasNext();) {
            String associationName = (String) i.next();
            c.setFetchMode(associationName,
              setFetchMode(fetch.get(associationName)));
        }
    }
    final String sort = (String)argMap.get(ARGUMENT_SORT);
    final String order = ORDER_DESC.equalsIgnoreCase(orderParam) ?
         ORDER_DESC : ORDER_ASC;
```

```
        final int max = maxParam == null ? -1 : maxParam.intValue();
        final int offset = offsetParam == null ? -1 : offsetParam.intValue();
        if(max > -1)
            c.setMaxResults(max);
        if(offset > -1)
            c.setFirstResult(offset);
        if(sort != null) {
            boolean ignoreCase = true;
            Object caseArg = argMap.get(ARGUMENT_IGNORE_CASE);
            if(caseArg instanceof Boolean) {
                ignoreCase = ((Boolean)caseArg).booleanValue();
            }
            if(ORDER_DESC.equals(order)) {
                c.addOrder( ignoreCase ?
                  Order.desc(sort).ignoreCase() : Order.desc(sort));
            }
            else {
                c.addOrder( ignoreCase ?
                  Order.asc(sort).ignoreCase() : Order.asc(sort) );
            }
        }

    }
```

可以重载 populateArgumentsForCriteria 方法，加入 class 参数以判断 Domain 类是否配置了 cache，再根据 Map 中 cache 键的取值，为 Criteria 设置 cacheable 属性即可。

```
    public static void populateArgumentsForCriteria(Criteria c, Map argMap,
        Class domainClazz) {
        populateArgumentsForCriteria( c,  argMap);
        if(isDomainCacheable(domainClazz)) {
            if(argMap.containsKey("cache")) {
                Boolean isCacheable =
                    (Boolean)converter.convertIfNecessary(
                        argMap.get("cache"),Boolean.class);
                        c.setCacheable(isCacheable.booleanValue());
            }
        } else {
            c.setCacheable(false);
        }
    }
```

问题还没有完，由于并没有修改原有的 populateArgumentsForCriteria 方法，而只是添加了新的方法，因此，还需要修改具体的查询方法（如 find*、findAll*、list 等方法）。

GORM 中封装的众多数据库访问方法，可以在 org.codehaus.groovy.grails.orm.hibernate.

metaclass 包中找到。每个非抽象类都对应一个具体的方法，如 FindAllByPersistentMethod 类实现了 findAllBy*方法，ListPersistentMethod 类实现了 list 方法，等等。

这些类重写（override）了 AbstractStaticPersistentMethod 类的 protected abstract Object doInvokeInternal(Class, String, Object[])方法，并且在该方法中实现具体的查询过程。以 ListPersistentMethod 类为例：

```
protected Object doInvokeInternal(final Class clazz, String methodName,
            final Object[] arguments) {
    // if there are no arguments list all
    if(arguments == null || arguments.length == 0) {
        return super.getHibernateTemplate().loadAll(clazz);
    }
    // otherwise retrieve the max argument
    else {
    // and list up to the max
        return super.getHibernateTemplate().executeFind(
            new HibernateCallback() {
                public Object doInHibernate(Session session) throws
                        HibernateException, SQLException {
                    Criteria c =  session.createCriteria(clazz);
                    if(arguments.length > 0) {
                        if(arguments[0] instanceof Map) {
                            Map argMap = (Map)arguments[0];
    GrailsHibernateUtil.populateArgumentsForCriteria(c,argMap);//@
                        }
                    }
                    return c.list();
                }
            }
        );
    }
}
```

其中“@”所在的行调用了不包含 class 参数的 populateArgumentsForCriteria 方法，需要用刚才新创建的方法取代它：

```
GrailsHibernateUtil.populateArgumentsForCriteria(c,argMap, clazz);
```

这里的 clazz 参数，恰好是 Domain 类的 class 属性。因而，实现了为 list 方法添加配置 Query Cache。对其他 GORM 查询方法的改造，在原理上和 list 方法是一样的。

修改完成后，重新编译 Grails。在项目中修改配置文件 grails-app\conf\Config.groovy 以开启 Hibernate 对 cache 的日志，修改方式为：

```
hibernate{ cache="debug"}
```

这样，就可以测试开启 Query Cache 的效果了。

19.4 本章小结

本章对 Grails 的部分源代码进行了简单的剖析，通过对源代码的阅读和修改，读者可以更好地学习和理解 Grails 技术。本章希望起到一个抛砖引玉的作用，期待聪明的读者发掘出更多更有趣的秘密。

第20章 未来 Grails 版本的新特性

本书介绍的 Grails 基于当前最新的稳定版本，即 1.0.4 版。下一个重要的 release 版本是 1.1[1]，有很多明显的改进和增强。

20.1 GORM 的新特性

20.1.1 更多的 GORM 事件

在 1.1 版中，GORM 增加了 afterInsert、afterUpdate 和 afterDelete 这 3 个事件，从而得到了一个更完整的事件模型。各个事件的触发时机如表 20-1 所示。

表 20-1　GORM 中的事件

事件触发时机	实例

插入前触发	def beforeInsert = {
	//do something before insert
	}

更新前触发	def beforeUpdate = {
	//do something before update
	}

删除前触发	def beforeDelete = {
	//do something before delete
	}

载入后触发	def onLoad = {
	//do something after loaded
	}

插入后触发	**def afterInsert = {
	//do something after insert
	}**

更新后触发	**def afterUpdate = {
	//do something after update
	}**

删除后触发	**def afterDelete = {
	//do something after loaded
	}**

[1] 当前的 Grails 1.1 还处于 Beta 2 测试阶段。

20.1.2　映射基本类型的集合

GORM 现在支持使用关联表（join table）对基本类型（如 String、Integer 等）进行映射。例如：

```
class Person {
   static hasMany = [nicknames:String]
}
```

则此时数据库表的 DDL 为：

```
CREATE TABLE 'person' (
  'id' bigint(20) NOT NULL AUTO_INCREMENT,
  'version' bigint(20) NOT NULL,
  PRIMARY KEY ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE 'person nicknames' (
  'person_id' bigint(20) DEFAULT NULL,
  'nicknames_string' varchar(255) DEFAULT NULL,
  KEY 'FK572C955BC12321BA' ('person_id'),
  CONSTRAINT 'FK572C955BC12321BA' FOREIGN KEY ('person_id') REFERENCES
  'person' ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

显然，Grails 根据 Person 和 nickname 间的一对多关系，为 nickname 创建了 person_nicknames 表，并创建了外键约束。

20.1.3　对 Domain 的只读访问

Grails 1.1 新增了 read 方法，用于实现获取只读 Domain 类实例，例如：

```
def book = Book.read(1)
```

20.1.4　定义默认排序字段

第 7 章中，设计了 Cart 和 LineItem 两个 Domain 类用于描述购物车。访问购物车中内容的代码如下：

```
cart.lineItems
```

cart.lineItems 这样的写法虽然简单却有一定的不足，那就是，无法指定 lineItems 的排序规则。Grails 1.1 提供了定义默认排序字段的方法，可以弥补这个不足。

Grails 1.1 现在可以在类一级或者关联一级指定默认排序字段，下面的代码展示了如何在类级别定义默认排序字段：

```
class LineItem {
Goods goods
    int itemNumber = 1
    static belongsTo = [cart:Cart]
    static constraints = {
        itemNumber(min:1)
    }
    static mapping = {
        sort "goods.id"
    }
}
```

此时，如果调用 list 方法对 LineItem 进行查询，则返回：

```
Hibernate:
   select
      this .id as id4 0 ,
      this_.version as version4_0_,
      this_.cart_id as cart3_4_0_,
      this_.goods_id as goods4_4_0_,
      this_.item_number as item5_4_0_
   from
      line_item this_
   order by
      this_.goods_id asc
```

但此时调用 cart.lineItems 仍然没有对 LineItem 进行排序查询。若要在访问 cart.lineItems 时进行排序，则需要在关系级别定义默认排序字段：

```
class Cart {
   static hasMany = [lineItems:Book]
   static mapping = {
           books sort:"goods id"
   }
}
```

则此时生成的 SQL 为：

```
Hibernate:
   select
      lineitems0_.cart_id as cart3_1_,
      lineitems0 .id as id1 ,
      lineitems0_.id as id3_0_,
      lineitems0_.version as version3_0_,
```

```
        lineitems0_.cart_id as cart3_3_0_,
        lineitems0_.goods_id as goods4_3_0_,
        lineitems0 .item number as item5 3 0
    from
        line_item lineitems0_
    where
        lineitems0_.cart_id=?
    order by
        lineitems0_.goods_id
```

值得注意的是，排序的字段需要写为 goods_id 而不是 goods.id，这可能是 Beta 版的 bug。

20.1.5 改进的 findBy

Grails 1.1 中的 findBy 方法，支持 inList 查询条件，支持 Query Cache，支持悲观锁。

```
//使用 inList 查询 author in ['Dierk Koenig', 'Graeme Rocher']的 Book
def groovyBooks = Book.findByAuthorInList(['Dierk Koenig', 'Graeme Rocher'])

//在查询中开启 Query Cache
def books = Book.findByTitle("Groovy in Action", [cache:true] )

//在查询过程中强制使用悲观锁
def books = Book.findByTitle("Groovy in Action", [lock:true] )
```

20.2 对插件系统的改进

Grails 1.0.4 中，需要为每个应用单独安装插件，到了 Grails 1.1 中，可以一次性为所有的应用安装插件，即全局插件：

```
>grails install-plugin webtest -global
```

20.3 数据绑定

Grails 1.1 对表单提交时的数据绑定也进行了改进。具体表现为 Domain 类的 properties 属性，首先回忆一下 1.1 版之前的 properties 属性的使用：

```
def goods = new Goods()
goods.properties = params
```

上面的写法存在一定的风险。假如希望提交的表单不包含 goods 的全部属性，则理论

上提交后的 params 中也不应包含 goods 的全部属性。然而，恶意的用户可以伪造提交的表单内容，在其中包含一些本不希望更新的属性，从而达到恶意更新的目的。

Grails 1.1 中，可以为 properties 指定部分需要更新的属性，例如：

```
person.properties["firstName","lastName"] = params
```


20.4 在 GSP 中使用 JSP 的标签

Grails 1.1 可以在 GSP 中使用 JSP 标签，这对实现历史遗产的支持，有着重大的意义，下面的例子展示了如何在 GSP 中调用 JSTL 的 formatNumber 标签：

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<fmt:formatNumber value="${10}" pattern=".00"/>
```

不仅如此，在 GSP 中，还可以像调用普通方法那样调用 JSP 的标签，例如：

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
${fmt.formatNumber(value:10, pattern:".00")}
```

20.5 加密配置文件中的数据库密码

DataSource 中配置了数据库的密码。在 1.1 版之前，都是在 DataSource 中使用明文作为密码。然而，使用明文作密码，总是不够安全的。Grails 1.1 中，可以通过指定 Codec，实现在 DataSource 配置文件中使用密文作为数据库的密码：

```
dataSource {
      username = "foo"
      password = "438uodf9s872398783r"
      passwordEncryptionCodec="my.company.encryption.BlowfishCodec"
}
```

其中的 passwordEncryptionCodec 属性指定了用于对密文密码（438uodf9s872398783r）进行解密的类。其原理与第 6 章介绍的 Codec 类是一样的。

20.6 本章小结

本章介绍了 Grails 1.1 的一些新特性，当前 Grails 1.1 还处于 Beta2 的状态，部分新功能还不够成熟。但从现在看到的这些改进，能看到 G2One 小组对 Grails 所付出的努力，也更有理由相信 Grails 1.1 的明天会更好！

参 考 文 献

[1] http://www.infoq.com/cn/news/2007/07/grails-misconceptions.

[2] http://grails.org/Success+Stories.

[3] http://baike.baidu.com/view/25258.htm.

[4] http://www.hibernate.org/hib_docs/reference/en/html/mapping.html#mappingdeclaration-id-generator.

[5] http://www.hibernate.org/hib_docs/reference/en/html/queryhql.html.

[6] http://jira.codehaus.org/browse/GRAILS-2899.

[7] http://static.springframework.org/spring/docs/2.5.x/reference/index.html.

[8] 深入浅出 REST，http://www.infoq.com/cn/articles/rest-architecure.

[9] 跨越边界: REST on Rails, http://www.ibm.com/developerworks/cn/java/j-cb08016/.

[10] http://www.grails.org/Plugins.

[11] Groovy 无痛 AOP 之旅，http://www.infoq.com/cn/articles/aop-with-groovy.

[12] Painless AOP with Groovy，http://www.infoq.com/articles/aop-with-groovy.

[13] http://ant.apache.org/.

[14] http://gant.codehaus.org/.

[15] http://jira.codehaus.org/browse/GRAILS-2899.

索　　引

本索引约定如下：词条按英文字母顺序排列（如果是中文，则以中文对应的汉语拼音为序）；索引中列出了部分词条的中英文对照，但对一些约定俗成的专用英文词条或缩写仅列出英文词条；词条所在页码指该词条在文中首次出现（或特别定义的地方）的页码。